Stackless Processing of Streamed Trees

Corentin Barloy
ENS de Paris, France
cbarloy@clipper.ens.fr

Filip Murlak
University of Warsaw, Poland
fmurlak@mimuw.edu.pl

Charles Paperman
Université de Lille & INRIA, France
charles.paperman@univ-lille.fr

ABSTRACT

Processing tree-structured data in the streaming model is a challenge: capturing regular properties of streamed trees by means of a stack is costly in memory, but falling back to finite-state automata drastically limits the computational power. We propose an intermediate stackless model based on register automata equipped with a single counter, used to maintain the current depth in the tree. We explore the power of this model to validate and query streamed trees. Our main result is an effective characterization of regular path queries (RPQs) that can be evaluated stacklessly—with and without registers. In particular, we confirm the conjectured characterization of tree languages defined by DTDs that are recognizable without registers, by Segoufin and Vianu (2002), in the special case of tree languages defined by means of an RPQ.

KEYWORDS

streaming, querying, XML, JSON, automata, weak validation

ACM Reference Format:
Corentin Barloy, Filip Murlak, and Charles Paperman. 2020. Stackless Processing of Streamed Trees. In *Proceedings of Principles of Database Systems (PODS'21).* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

While graph is the new black, tree-structured data has not vanished. It is used both as a serialization format (Wikipedia, Wikidata, DBLP) and as an exchange format (WSDL and SOAP rely on XML, the more recent GraphQL prefers JSON). Querying and validation of tree-structured data continue to be both vital and challenging tasks in data management. Particularly so, when documents grow too large to fit in memory, and it is time to switch to streaming; that is, to read the document sequentially, maintaining a concise internal representation sufficient for the realized task.

According to Palkar et al. [18], exploratory big-data applications running over data in a semi-structured format, like JSON, can spend 80-90% of their execution time simply parsing the data. Performance improvements often rely on clever ways to reduce the cost of parsing. In systems research, two main strategies have been proposed. The first one relies on SAX (*Simple API for XML*) parsers: it outsources parsing to the API and deals only with the resulting events [11, 26]. This allows to factor out the cost of parsing, and may lead to significant performance gains when multiple queries are executed over the same document [26]. The second approach is to perform parsing and query execution simultaneously, applying push-down automata as the computation model [17], in the hope that the acquired semantic information would help reduce the cost of parsing. When a single query is executed over a huge document, this may also be highly beneficial [6].

The theoretical take on alleviating the cost of parsing is more radical: since it is so costly, let us assume that it has been already done for us and the input stream is guaranteed to be a well-formed document. This may be the case, for instance, if we trust the source of the document or if we have already processed the document for other purposes. Can this assumption help process the document more efficiently? This setting was introduced as *weak validation* in the seminal work of Segoufin and Vianu [25] on validating a streamed XML document against a DTD by means of a finite automaton. Despite the significant progress made in the initial paper and in the follow-up work [1, 5, 24], the general problem of deciding whether weak validation against a given regular tree language is feasible, remains open. Incidentally, this question is a special—but disturbingly generic—case of an undecidable separation problem [13].

A recent trend in data processing is to use hardware acceleration to exploit *local parallelism*. Most modern CPU architectures offer SIMD (*single instruction multiple data*) instructions, allowing to perform the same operation on multiple data points in one CPU cycle, leading to what is known as the *vectorization* of computation. Vectorization in used routinely in data-intensive applications like multimedia processing [23] or deep learning [8, 27], and is finding its way to data management, particularly in the sub-field of in-memory databases [22, 32]. Relevant examples from a related field are the performant regular expression engine *Hyperscan* [29] and the competitive engine of the RUST language [10], both relying crucially on vectorization. In the context of streaming processing of tree-structured data, an early work on *parabix* by Cameron et al. studies the use of SIMD instructions to accelerate XML parsing [4]. More recently, Langdale and Lemire illustrate how the performance of JSON parsers could be vastly improved by using vectorization [14]. Their experiments confirm that the cost of parsing is a large fraction of the total cost of query execution, matching the performance loss with respect to regular expression matching. To get a better feeling of the room for improvement, let us look at some numbers: the experiments had different setups, but the orders of magnitude are still of interest. The standard C function MEMCHR scans memory to find the first occurrence of a given byte; it has been hand-optimized for various architectures and can be assumed to display the best performance one could hope for in a streaming task. On a standard laptop computer, it easily reaches 20Gb/s. The *Hyperscan* regular expression engine reaches performance of 10Gb/s [29]. Langdale et al. get up to 3Gb/s when parsing JSON files and selecting some nodes, but selecting alone reaches 10Gb/s [14].

PODS'21, June 20-June 25, 2021, Xi'an, Shaanxi, China

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Palkar et al. explicitly put the blame on the incompatibility of pushdown automata and vectorization [18].

To some extent this is explained by theory. An abstract model of exploiting local parallelism in streaming algorithms was proposed in [15]: the stream is read in blocks, each block is processed by a fixed boolean circuit, and the result is fed back to the circuit together with the next block of the stream. The degree of local parallelism of a language is measured by the complexity of the circuit needed to recognize the language in the above model: the higher the complexity, the less local parallelism. As shown in the paper, the degree of local parallelism of regular languages matches their classical circuit complexity, and it is plausible that the situation is similar for larger classes of languages. Assuming this is the case, successful vectorization of XML or JSON parsers might be more tricky than for regular expression engines: Dyck languages (well-formed multi-bracket expressions) are TC^0-complete [2], but while regular languages may have even higher complexity, the ones appearing in benchmarks are typically much simpler (for instance, all examples in [10] and [31] are in AC^0).

All this evidences that stack-based computation is troublesome. At the same time falling back to finite automata severely limits expressivity, as revealed by the necessary conditions discovered by Segoufin and Vianu [25]. As a middle ground, we propose to relax the computational model just so. We allow one counter for maintaining the current depth in the document, and registers for storing the current depth to be compared with the depths of later tags. In the resulting model, dubbed *depth-register automaton*, transitions are performed at a very low CPU cost with almost no external memory access. The latter depends on the number of registers; if the number is low enough, it is even possible to keep all the values within the CPU's registers and not use external memory at all. Unlike pushdown automata, the model appears amenable to vectorization and may be hoped to achieve high throughputs, but a systematic study of this aspect is a matter of future work.

We apply the proposed setting (predominantly) to querying, which—to the best of our knowledge—has not been studied before from this angle. After a preliminary expressivity study, we embark on characterizing node-selecting queries that can be realized in our model. Our first main result is an effective characterization of regular path queries (RPQs) that can be realized with depth-register automata. As a by-product, we reveal a connection with the languages of trees in which some (resp. each) leaf is selected by the RPQ. Our second main result is an analogous characterization for finite automata. The conditions for the unary query and the two associated tree languages do not conincide any more, but are elegantly related: the RPQ can be realized iff both languages can be (weakly) recognized. The setting of the second main result is exactly that of Segoufin and Vianu, and it allows to make some progress towards solving the original weak validation problem. We develop our results for the XML encoding of trees, but they adapt smoothly to the less verbose JSON-style encoding.

Organization of the paper. Section 2 introduces the computation model and gives a preliminary expressivity study. Section 3 establishes the characterization theorems. Section 4 explores the connection with the weak validation problem, explains the adaptation to JSON-style encoding, and points out key open problems.

2 COMPUTATIONAL MODEL

We model tree-structured data as ordered unranked finite trees whose nodes are labelled with symbols from a finite alphabet Γ. We refer to them simply as *trees over* Γ. An *immediate subtree* of tree T is a subtree rooted at a child of the root of T. A *tree language over* Γ is a set of trees over Γ. If L is a language of trees (or words) over Γ, we write L^{c} for the complement of this language.

The *markup encoding* represents trees over Γ as words over the alphabet $\Gamma \cup \bar{\Gamma}$, where $\bar{\Gamma} = \{\bar{a} \mid a \in \Gamma\}$. In the context of the encoding, the elements of Γ and $\bar{\Gamma}$ are referred to as *opening and closing tags*, respectively. If T is a tree whose root is labelled with a and whose immediate subtrees are $T_1, T_2, \ldots, T_n$, then

$$\langle T \rangle = a \cdot \langle T_1 \rangle \cdot \langle T_2 \rangle \cdot \cdots \cdot \langle T_n \rangle \cdot \bar{a}\,.$$

For example, $aa\bar{a}c\bar{c}\bar{a}$ encodes the tree with a-labelled root whose first child has label a and second child has label c. If L is a tree language over Γ, we let $\langle L \rangle = \{\langle T \rangle \mid T \in L\} \subseteq (\Gamma \cup \bar{\Gamma})^*$.

2.1 Depth-register automata

Under the markup encoding, finite automata are unable to check even the simplest properties of the input document: for instance, determining if one marked node is a child, descendant, or sibling of another marked node requires a stack—or at least a counter, used to compare depths of nodes. Realizing multiple such tasks simultaneously seems to lead to multi-counter automata, which are notoriously hard to analyze. We take a different path: we allow only one counter, used exclusively to maintain the current depth in the tree, but additionally equip the automaton with a bounded number of registers, which can be used to store depths of critical nodes, and compare them later with the current depth. To keep our automata executable efficiently, we assume that they are deterministic. Thus we arrive at *deterministic input-driven 1-counter automata with registers*. 'Input-driven' is the standard terminology for counters or stacks that evolve independently of the state [7, 28]. Here it means that the counter increases by one with each opening tag read, and decreases by one with each closing tag read; such automata (without registers) are also called *visibly counter automata* [1]. Importantly, the only tests allowed on the values stored in registers are order comparisons with the current depth. We shall refer to such devices as *depth-register automata*. A formal definition follows.

Definition 2.1. A *depth-register automaton* $\mathcal{A}$ is a tuple

$$(\Gamma, Q, q_{init}, F, \Xi, \delta)\,,$$

where Γ is a finite alphabet, Q is a finite set of states, $q_{init} \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting (final) states, Ξ is a finite set of registers, and

$$\delta : Q \times (\Gamma \cup \bar{\Gamma}) \times 2^{\Xi} \times 2^{\Xi} \to 2^{\Xi} \times Q$$

is the transition function.

A *configuration* of $\mathcal{A}$ is a tuple $(q, d, \eta) \in Q \times \mathbb{Z} \times \mathbb{Z}^{\Xi}$, whose components specify the state, the current depth, and the values stored in the registers, respectively. We call a configuration (q, d, η) *accepting* if $q \in F$. The *initial configuration* is $c_{init} = (q_{init}, 0, \eta_{init})$ where $\eta_{init}(\xi) = 0$ for all $\xi \in \Xi$.

The *run of $\mathcal{A}$ over a word $a_1 a_2 \dots a_n \in (\Gamma \cup \bar{\Gamma})^*$ from a configuration* (q_0, d_0, η_0) is the unique sequence of configurations

$$(q_0, d_0, \eta_0)(q_1, d_1, \eta_1) \dots (q_n, d_n, \eta_n) \in \left(Q \times \mathbb{Z} \times \mathbb{Z}^{\Xi}\right)^*$$

such that for each $i \in \{1, 2, \dots, n\}$, there exists $Y_i \subseteq \Xi$ such that

- $d_i = \begin{cases} d_{i-1} + 1 & \text{if } a_i \in \Gamma, \\ d_{i-1} - 1 & \text{if } a_i \in \bar{\Gamma}; \end{cases}$
- $\delta(q_{i-1}, a_i, X_i^{\leq}, X_i^{\geq}) = \left(Y_i, q_i\right)$ where
$$X_i^{\leq} = \{\xi \in \Xi \mid \eta_{i-1}(\xi) \leq d_i\},$$
$$X_i^{\geq} = \{\xi \in \Xi \mid \eta_{i-1}(\xi) \geq d_i\};$$
- for each $\xi \in \Xi$,
$$\eta_i(\xi) = \begin{cases} d_i & \text{if } \xi \in Y_i, \\ \eta_{i-1}(\xi) & \text{if } \xi \notin Y_i. \end{cases}$$

We write $c \cdot w$ for the last configuration of the run on w from c. If $c \cdot w = c'$, we also write $c \xrightarrow{w} c'$. By the *run of $\mathcal{A}$ on w* we understand the run on w from c_{init}. We say that w is *accepted* by $\mathcal{A}$ if $c_{init} \cdot w$ is accepting. The *language recognized by $\mathcal{A}$* is the set of words accepted by $\mathcal{A}$.

Depth-register automata without registers (that is, with $\Xi = \emptyset$) are a notational variant of deterministic finite automata over the alphabet $\Gamma \cup \bar{\Gamma}$. For such automata we streamline the notation introduced in Definition 2.1 by dropping the ingredients associated with Ξ. In particular, we use states instead of configurations, and write $q \cdot w = q'$ and $q \xrightarrow{w} q'$. The same notation will be applied to finite automata over Γ, not only over $\Gamma \cup \bar{\Gamma}$.

We shall give examples of depth-register automata in Section 2.2 (Examples 2.2, 2.5 and 2.6), once we have made precise how they are used to recognize tree languages.

To conclude the discussion of the automata model, let us point out that the kind of tests allowed on registers is a natural parameter of the definition. For instance, one could allow testing if the current depth differs from the content of a given register by a specified constant; this kind of test can be simulated in our model at the cost of using additional registers. An interesting proper extension is to allow semilinear conditions, like testing equality modulo a specified constant. Finally, forsaking any hope of decidability of emptiness (which might be tolerable), one could go up to full arithmetics. Owing to their determinizm, depth-register automata in all these variants would be efficiently executable in practice, using only a constant number of variables (possibly just CPU registers). Nevertheless, in this first study we stick to the minimalist approach.

2.2 Recognizing streamed tree languages

A tree language L over Γ is *recognized (under the markup encoding)* by an automaton $\mathcal{A}$ over the alphabet $\Gamma \cup \bar{\Gamma}$ if for each tree t over Γ it holds that $\mathcal{A}$ accepts $\langle t \rangle$ iff $t \in L$. Equivalently, L is recognized by $\mathcal{A}$ if the language of words accepted by $\mathcal{A}$ separates $\langle L \rangle$ from $\langle L^c \rangle$. A tree language is *stackless* if it is recognized by a depth-register automaton, and *registerless* if it is recognized by a finite automaton.

Note that the automaton is allowed to accept or reject *invalid encodings*; that is, elements of $\left(\Gamma \cup \bar{\Gamma}\right)^* \setminus \left(\langle L \rangle \cup \langle L^c \rangle\right)$. Requiring that all invalid encodings be rejected would lead to $\langle L \rangle \subseteq (\Gamma \cup \bar{\Gamma})^*$ being recognized by a finite (resp. depth-register) automaton, which is a much stronger property. In particular, the assumption that $\langle L \rangle$ is recognized by a finite automaton is prohibitively strong, as it implies that the depth of trees in L is bounded [25]. In contrast, a registerless tree language may easily contain trees of unbounded depth: a very simple example is the set of trees with at least one a-labelled node, which can be recognized (under the markup encoding) by a finite automaton that moves to an all-accepting sink state upon reading the opening tag a for the first time.

Registerless tree languages are regular, because a tree automaton can simulate the run of a finite automaton over the encoding of the tree. Stackless tree languages, in contrast, need not be regular.

Example 2.2. The set of trees over the alphabet $\{a, b\}$ in which all a-labelled nodes are at the same depth, can be recognized by a depth-register automaton. The first time the automaton sees a, it stores the current depth in its only register. Then, every time it sees a it checks if the current depth is equal to the stored value, and if it is not, it moves to a rejecting sink state.

Regularity can be enforced by applying a stack-like policy of using registers. We call a depth-register automaton *restricted* if each transition overwrites all stored values strictly greater than the current depth; that is, if $\delta(p, a, X^{\leq}, X^{\geq}) = (Y, q)$, then $X^{\geq} \setminus X^{\leq} \subseteq Y$.

Proposition 2.3. *Restricted depth-register automata recognize regular tree languages.*

We conjecture that restricted depth-register automata recognize all regular stackless tree languages, but it is conceivable that they do not. This is why we work with the unrestricted model and prove (potentially) stronger inexpressibility results. We stress, however, that all depth-register automata we construct are restricted. In particular, the characterization in Theorem 3.1 is identical for the restricted model, backing up the conjecture.

Regardless of the restriction, stackless tree languages retein the usual closure properties or regular languages.

Lemma 2.4. *The classes of registerless and stackless tree languages are both closed under intersection, union, and complementation.*

How far do stackless tree languages go beyond registerless? As first examples, let us see how depth-register automata can deal with sequences of siblings and the descendent relation.

Example 2.5. Consider a regular language $L \subseteq \Gamma^*$ and the set HL of trees over Γ such that the sequence of labels read from the children of the root forms a word in L. Depending on L, the tree language HL may be registerless or not. For instance, for $L = \Gamma^* a \Gamma^*$, HL is not registerless, because a finite automaton cannot determine whether the current tag with label a belongs to a child of the root. This can be shown easily by pumping, but it also follows from our general characterization result, Theorem 3.2 (1), applied to the set of trees that contain a branch labelled by a word from $\Gamma a \Gamma^*$. In contrast, HL is stackless for all regular L. Indeed, after reading the first tag (which must be an opening tag in a valid encoding), the automaton stores the current depth (which is 1) in its only register, and then simulates the finite automaton recognizing L over all closing tags for which the current depth is equal to the value stored in the register. This is correct, because in each valid encoding all closing tags with current depth 1 belong to the children of the root.

Example 2.6. Consider the set of trees over the alphabet $\{a, b, c\}$ where the first a-labelled node (in the document order) has a b-labelled descendent. To recognize this language, the automaton should read the input word until it sees a, load the current depth to its only register, and accept iff it sees the letter b before the current depth drops strictly below the stored value (this will indicate, that the corresponding closing tag has been read). Now, consider the set of trees over $\{a, b, c\}$ where *some* a-labelled node has a b-labelled descendant. It suffices to test this property for minimal a-labelled nodes (that is, those without a-labelled ancestors): if a node has a b-labelled descendent, so do all its ancestors. Hence, to recognize the described language it suffices to run the automaton described above in a loop, returning to the initial state whenever the current depth drops strictly below the stored value, until it accepts.

The main weakness of depth-register automata when applied to processing trees is their limited ability to handle the child relation, as revealed by the following example.

Example 2.7. Consider the language of trees over the alphabet $\{a, b, c\}$ where some a-labelled node has a b-labelled child. It might appear that this language is stackless because it is easy to identify an a-labelled node and a single register is sufficient to identify the tags of its children in the encoding. Indeed, this idea can be used to recognize the language of trees where some *minimal* a-labelled node has a b-labelled child, just like we did for b-labelled descendents in Example 2.6. Without the minimality assumption, however, the subautomaton searching for b-labelled children needs to be relaunched whenever the opening tag a is read, which may well happen before the previous instance of the subautomaton terminates. Each launch requires a new register to store the return point. Because the input tree may contain arbitrarily long chains of a-labelled nodes, this does not seem feasible with any fixed number of registers. That it is indeed infeasible follows from the general characterization result (Theorem 3.1) we establish in Section 3.

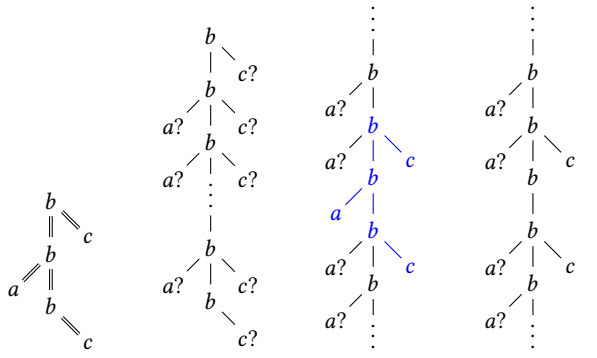
The method from Example 2.6 can be extended to test the existence of multiple nodes with specified labels and descendent relationships between them. By a *descendent pattern* we shall understand a finite tree over Γ. A tree T *contains* a descendent pattern π if there exists a matching function h that maps nodes of π to nodes of T such that for all nodes u, v of π:

- the label of u coincides with the label of $h(u)$;
- if v is a child of u, then $h(v)$ is a descendent of $h(u)$.

Proposition 2.8. *For each descendent pattern π, the set of trees containing π is stackless.*

Proof. By a slight abuse of the definition of depth-register automata, we shall allow automata that can stop; that is, in some configurations there may be no transition to take. We prove by induction on the height of π that there is an automaton $\mathcal{A}_\pi$ that recognizes trees that contain π and stops upon reading the closing tag corresponding to the first opening tag of its input.

If π consists of a single node, the automaton loads into its only register the current depth before reading any tags, scans the input until the current depth again becomes equal to the stored value. Then it moves to a state without outgoing transitions that is accepting or not, depending on whether the automaton has detected a tag with the label from the root of π or not.



(a) Pattern π. **(b) A 'schema'.** **(c) Match.** **(d) No match.**

Figure 1: Strict descendent patterns are not stackless.

Suppose that the root of π has some children. By the inductive hypothesis, there is an automaton $\mathcal{A}_{\pi'}$ for each descendent pattern π' corresponding to an immediate subtree of π. Let $\mathcal{A}$ be the synchronous product of all these automata, recognizing the intersection of the languages recognized by its components. Like in Example 2.6, we can assume that the root of π is matched to a minimal element with the desired label. The automaton $\mathcal{A}_\pi$ loads the current depth before reading any tags into its first register, and then processes the input looking for the first opening tag with the same label as the root of π. If $\mathcal{A}_\pi$ does not see one before the current depth is again equal to the stored value, it rejects. If it does find one, it calls the automaton $\mathcal{A}$ using a set of registers excluding the first one and waits until $\mathcal{A}$ stops. If $\mathcal{A}$ accepts, $\mathcal{A}_\pi$ waits until the current depth becomes equal to the value stored in the first register, and accepts. If $\mathcal{A}$ rejects, $\mathcal{A}_\pi$ moves on to the next opening tag with the same label as the root of π. □

While the class of stackless tree languages is closed under complement by Lemma 2.4, it does not follow that we can handle negative information just as well as positive. We say that a tree T *strictly contains* a descendent pattern π if T contains π and the matching function h additionally satisfies the condition

- if v is not a descendent of u, then $h(v)$ is not a descendent of $h(u)$.

The following example shows that Proposition 2.8 does not extend to this stronger notion.

Example 2.9. Consider the pattern π shown in Fig. 1a. As is customary, we use double edges to indicate descendent relationships between nodes. Suppose that the languages of trees strictly containing π is recognized by a depth-register automaton $\mathcal{B}$ with m states and ℓ registers. We shall analyze the behaviour of the automaton $\mathcal{B}$ over trees conforming to the 'schema' shown in Fig. 1b, which for each $n > 2$ defines the set K_n of trees that have the main branch labelled by the word b^n, and additionally each b-labelled node may have a c-labelled child to the right of the main branch, and each *internal* b-labelled node may have an a-labelled child to the left of the main branch. For a tree T like this, let w_T be the prefix of $\langle T \rangle$

ending at the opening tag of the deepest b-labelled node. Let c_{init} be the initial configuration of $\mathcal{B}$.

For $T \in K_n$, we have that $c_{init} \cdot w_T = (q, n, \eta)$ for some state q and some $\eta : \Xi \to \{0, 1, \ldots, n\}$. That is, $m \cdot (n+1)^\ell$ configurations are possible. But there are 2^{n-2} ways to choose which b-labelled non-leaf nodes have an a-labelled child, so $\left|\{w_T \mid T \in K_n\}\right| = 2^{n-2}$. Consequently, for sufficiently large n, there exist two different words u and v in $\{w_T \mid T \in K_n\}$, such that $c_{init} \cdot u = c_{init} \cdot v$. Because $u \neq v$, there exists $i \in \{2, 3, \ldots, n-1\}$ such that for all $S, T \in K_n$, if $u = w_S$ and $v = w_T$, then the ith b-labelled node has an a-labelled child in S iff it does not have one in T. Let us choose S and T such that in both of them, the $(i-1)$st and the $(i+1)$st b-labelled node has a c-labelled child and there are no other c-labelled nodes, as shown in Figs. 1c and 1d. Clearly, the tree in Fig. 1c strictly contains π. It is not difficult to verify that the one in Fig. 1c does not. However, from the definition of S and T it follows that $\langle S \rangle = uw'$ and $\langle T \rangle = vw'$ for some w', and because $c_{init} \cdot uw' = c_{init} \cdot vw'$, we conclude that S and T are indistinguishable to $\mathcal{B}$.

Finally, let us point out that the ability to deal with sequences of siblings, demonstrated in Example 2.5, is limited to nodes that are close to the root. The following example shows why.

Example 2.10. Even a finite automaton can check if the streamed tree contains two consecutive siblings with labels a and b: it suffices to check if the read encoding contains the closing tag $\bar{a}$ followed immediately by the opening tag b. Consider, however, the set of trees that contain three consecutive siblings with labels a, b, c. Arguing like in Example 2.9 one can show that this language is not stackless. Dropping the assumption that the siblings are consecutive, or even that they are ordered as written, does not affect the argument.

Thus, depth-register automata are able to express involved global properties of trees (Proposition 2.8), far out of reach of finite automata, yet they cannot handle many properties that appear local but lose their locality when seen as properties of the encodings (Examples 2.7 and 2.10). Characterizing stackless tree languages seems to be challenging, but in Section 3 we solve the special case of tree languages defined in terms of properties of branches.

2.3 Querying streamed trees

So far we used automata as acceptors, defining languages of trees. However, we can also use them as node selectors, defining queries over trees. By a *query Q of arity k* we mean a function mapping each tree T to a set $Q(T)$ of k-tuples of nodes of T. In the streaming setting, higher-arity queries are problematic because a streaming algorithm using memory of size $f(n)$ over inputs of length n cannot return asymptotically more than $f(n) \cdot n$ answers. This means that handling even very simple queries of arity larger than one in sublinear memory is impossible without compromising the semantics by applying restrictive *selection strategies* [9, 30] or heuristics like *load shedding* [12]. Moreover, popular query languages for tree-structured data, like XPath or JSONPath, focus on unary queries. We shall do the same.

Implementations of unary queries over streamed trees come in two distinct flavours, corresponding to the two natural moments when one may wish the selected nodes to be returned: at the opening tag or at the closing tag. Accordingly, we say that an automaton $\mathcal{A}$ *pre-selects* (resp. *post-selects*) a node v of a tree T if $\mathcal{A}$ is in an accepting state directly after reading the opening (resp. closing) tag of v. Both approaches have their merits. Post-selection is more expressive, because it allows to explore the subtree rooted at the given node. Pre-selection gives more flexibility in the subsequent stages of processing, allowing to return the whole subtree rooted at the selected node without additional memory cost. In this work we focus on pre-selection, and leave post-selection for the future. We say that an automaton $\mathcal{A}$ realizes a unary query Q if for every tree T, $\mathcal{A}$ pre-selects exactly those nodes of T that belong to $Q(T)$. We call a unary query *stackless* (resp. *registerless*) if it can be realized by a depth-register automaton (resp. finite automaton).

Practical declarative query formalisms for tree-structured data, like XPath or JSONPath, treat the context of a node in a symmetric fashion, even if siblings are considered ordered. The streaming setup, on the other hand, is inherently asymmetric: siblings to the left of the node to be selected can be accessed freely, but there is no way to access those on the right. While there exist meaningful queries that could exploit access to the siblings on the left, in this work we abstract away from this aspect and focus on queries invariant under sibling order. A query Q is *invariant under sibling order* if for each bijection f between the nodes of a tree T and the nodes of a tree T' that preserves node labels and the child relation, it holds that $Q(T') = \{f(u) \mid u \in Q(T)\}$. Unary queries invariant under sibling order form a rich class and capture an important segment of user queries, including all vertical XPath queries, built up from vertical axes (child, descendent, parent, ancestor), label tests, and filters. We aim at understanding which of them can be implemented over streamed trees using finite or depth-register automata.

The scope of this task can be narrowed down quickly, as all stackless queries invariant under sibling order fall within a well-known class of queries. With each regular language $L \subseteq \Gamma^*$, we associate a unary query QL that selects all nodes v such that the path from the root to v is labelled by a word from L. We call queries of this form *regular path queries* (RPQs). They include all XPath queries built up from downward axes (child, descendent) and label tests, but not those using upward axes (parent, ancestor) or filters.

Proposition 2.11. *The class of stackless queries invariant under sibling order is contained in the class of RPQs.*

That is, if a unary query invariant under sibling order is not an RPQ, then it is not stackless either. In particular, vertical XPath queries cannot be realized by depth-register automata if they use upward axes or filters. Consequently, understanding which unary queries invariant under sibling order are stackless or registerless amounts to characterizing stackless and registerless queries among RPQs, which will be the focus of the remainder of this paper.

Example 2.12. Consider the following simple RPQs, expressed in XPath, JSONPath, and as regular expressions:

XPath	`/a//b`	`/a/b`	`//a//b`	`//a/b`
JSONPath	`$.a..b`	`$.a.b`	`$..a..b`	`$..a.b`
RegEx	$a\,\Gamma^* b$	$a\,b$	$\Gamma^* a\,\Gamma^* b$	$\Gamma^* a\,b$
Registerless?	✓	✗	✗	✗
Stackless?	✓	✓	✓	✗

The first one is registerless: the realizing finite automaton should check that the first opening tag has label a and then it should accept

at each opening tag with label b. On the other hand, the last RPQ cannot be realized even by a depth-register automaton, because letting this automaton loop in each accepting state we would obtain an automaton recognizing the language from Example 2.7. What about the remaining two RPQs? It will follow from our general characterization results (Theorems 3.1 and 3.2) that they are stackless, but not registerless.

From the perspective taken in this paper, RPQs and depth-register automata play asymmetric roles: RPQs represent user queries, and depth-register automata represent their implementations in the streaming setting. Accordingly, the fundamental question is which user queries can be implemented; that is, which RPQs are stackless. Nevertheless, one can also ask which stackless queries are RPQs. This appears challenging in general, but if the query is given as a restricted depth-register automaton, it is a pleasent exercise in automata theory, reminiscent of the characterization of tree languages recognizable by deterministic top-down automata [16].

Proposition 2.13. *It is decidable if the query realized by a given restricted depth-register automaton is an RPQ.*

Unlike in graph databases, where RPQs are viewed as binary queries selecting suitably connected pairs of nodes [3], in our setting RPQs are treated primarily as unary queries selecting nodes suitably connected to the root. But we can also treat them as boolean queries, defining sets of trees that contain a node—or a leaf—suitably connected to the root. The leaf variant will be instrumental in the characterization results of Section 3. We write EL for the set of trees that contain a branch labelled by a word from L, and AL for the set of trees with all branches labelled by words from L. Note that $(AL)^{\mathrm{c}} = E(L^{\mathrm{c}})$. Languages of the form AL can express useful and nontrivial schema restrictions, as they are able to specify which labels are allowed in the children of a node, depending on regular properties of the path from the root. This will allow us to shed more light on the framework of Segoufin and Vianu [25] in Section 4.1.

3 CHARACTERIZATION THEOREMS

The characterization theorems rely on four syntactic classes of regular languages: almost-reversible, hierarchically almost-reversible, E-flat, and A-flat (Definitions 3.4, 3.6 and 3.9). For now they can be treated as blackboxes, but let us highlight that their definitions are based on simple PTIME-testable properties of the minimal automaton, which makes the characterizations effective. Indeed, also the suitable automata for QL, AL, and EL can be computed in time polynomial in the size of the minimal automaton of L.

For each theorem we provide a proof outline explaining how to infer the theorem from the expressibility and inexpressibility results we establish in the remainder of this section.

Theorem 3.1. *For each regular language L, the following conditions are equivalent:*

(1) QL is a stackless unary query;
(2) EL is a stackless tree language;
(3) AL is a stackless tree language;
(4) L is hierarchically almost-reversible.

Proof outline. (1) implies (2) because an automaton $\mathcal{A}$ realizing QL can be easily turned into an automaton $\mathcal{A}'$ recognizing EL.

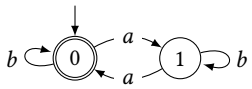


Figure 2: A reversible finite automaton.

$\mathcal{A}'$ behaves like $\mathcal{A}$, but it additionally remembers the previously read symbol; if the previous symbol was an opening tag, the state is accepting in $\mathcal{A}$, and the current letter is a closing tag, then $\mathcal{A}'$ moves to an all-accepting sink state. (2) implies (4) by Lemma 3.16, and (4) implies (1) by Lemma 3.8. This shows that (1), (2), and (4) are equivalent. It follows that (2) and (3) are equivalent, because $(AL)^{\mathrm{c}} = E(L^{\mathrm{c}})$, the class of stackless tree languages is closed under complementation, and, by Lemma 3.7, so is the class of hierarchically almost-reversible languages. □

In the registerless case the picture is more complicated, reflecting the inherent duality of tree languages of the form AL and EL.

Theorem 3.2. *Let L be a regular language.*

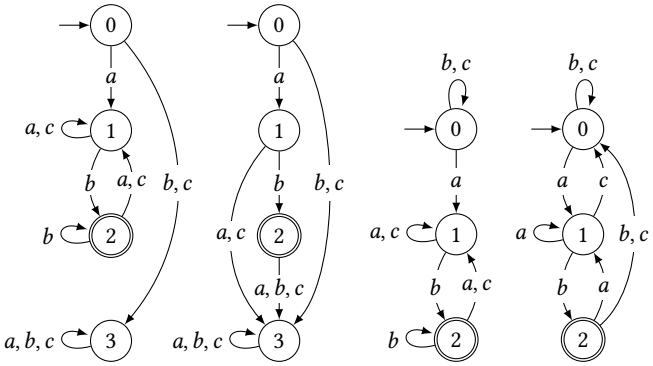
(1) EL is a registerless tree language iff L is E-flat.
(2) AL is a registerless tree language iff L is A-flat.
(3) The following conditions are equivalent:
  (a) QL is a registerless unary query;
  (b) EL and AL are registerless tree languages;
  (c) L is E-flat and A-flat;
  (d) L is almost-reversible.

Proof outline. (1) follows from Lemmas 3.11 and 3.12. (2) follows from (1) because: $(AL)^{\mathrm{c}} = E(L^{\mathrm{c}})$, the class of registerless tree languages is closed under complementation, and by Lemma 3.10, L is A-flat iff L^{c} is E-flat. For (3), we argue like in Theorem 3.1 that if QL is registerless, so is EL. Similarly, if QL is registerless, so is AL; the automaton $\mathcal{A}'$ is constructed dually: whenever it reads a closing tag immediately after an opening tag while being in a rejecting state of $\mathcal{A}$, it moves to the all-rejecting sink state $\bot$. Hence, (3a) implies (3b). (3b) is equivalent to (3c) by (1) and (2). (3c) is equivalent to (3d) by Lemma 3.10. (3d) implies (3a) by Lemma 3.5. □

We remark that Theorem 3.2 is fully compatible with the framework introduced by Segoufin and Vianu [25]; we discuss the connection in detail in Section 4.1.

3.1 Almost-reversibility

How does one go about evaluating an RPQ with a finite automaton reading the markup encoding of a tree? Over the *leftmost* branch this is easy: as long as only opening tags are read, we simulate the automaton underlying the RPQ over the labels in the tags and accept whenever the simulated automaton accepts. When the first closing tag appears, the simulated automaton should revert to the state before reading the corresponding opening tag. Our simulation could store a bounded suffix of the run of the simulated automaton, and use it when closing tags occur, but what shall we do when it is used up? This is clearly not a sustainable strategy. The task does become feasible if we assume that the previous state can be determined based on the current state and the last read letter. Automata that have this property are called *reversible*.



(a) $a\,\Gamma^* b$ **(b)** ab **(c)** $\Gamma^* a\,\Gamma^* b$ **(d)** $\Gamma^* ab$

Figure 3: Languages of increasing hardness over $\Gamma = \{a, b, c\}$.

Recall that in a deterministic automaton letters induce functions mapping states to states. A deterministic automaton is *reversible* if every letter induces an injective function (Fig. 2). Equivalently, one may assume that letters induce permutations of states, which implies that the monoid generated by these functions—with composition as the inner product—is a group. Reversibility can be studied as a separate notion upon extension to incomplete automata, where letters induce partial functions over states [20].

The simulation above captures RPQs given by reversible automata, but we can do a bit more. Consider the automata depicted in Fig. 3. None of them is reversible because the function induced by the letter a is not injective. However, as explained in Example 2.12, the automaton in Fig. 3a defines a registerless RPQ, while those in Figs. 3b to 3d do not. In order to capture registerless RPQs precisely, we carefully relax the notion of reversibility.

Unlike reversibility itself, its relaxed variant is dependent on which states are accepting. Let us fix a deterministic automaton $\mathcal{A}$. We say that states p and q are *equivalent* if for every word w, $p \cdot w \in F$ iff $q \cdot w \in F$. In a minimal automaton, equivalent states are equal. We say that states p and q are *almost equivalent* if for every non-empty word w, $p \cdot w \in F$ iff $q \cdot w \in F$. That is, non-empty words do not distinguish almost equivalent states; it follows immediately that after reading any letter the states become indistinguishable.

Lemma 3.3. *If states p and q are almost equivalent, then for each letter a, the states $p \cdot a$ and $q \cdot a$ are equivalent.*

We shall call a state p of automaton $\mathcal{A}$ *internal* if it is reachable from the initial state via a nonempty word. Note that if all states are reachable, only the initial state can be non-internal, and it happens only iff it has no incoming transitions.

Definition 3.4 (Almost-reversibility). We say that states p and q *meet in state r* if there exists a word u such that $p \cdot u = q \cdot u = r$; we say p and q *meet* if they meet in some state r. A deterministic automaton is *almost-reversible* if every two internal states that meet are almost equivalent. We call a regular language almost-reversible if its minimal automaton is almost-reversible.

As intended, the automaton in Fig. 3a is almost-reversible, while those in Figs. 3b to 3d are not.

Lemma 3.5. *If L is an almost-reversible language, then QL is a registerless query.*

Proof. Let $\mathcal{A}$ be the minimal automaton of L. The simulating automaton $\mathcal{B}$ will use the same states as $\mathcal{A}$ together with an additional rejecting sink state $\perp$; the initial state and the set of accepting states are also like in $\mathcal{A}$. When reading opening tags, $\mathcal{B}$ follows the transition relation of $\mathcal{A}$. Upon reading a closing tag $\bar{a}$ in a state p, $\mathcal{B}$ moves to some internal p' in $\mathcal{A}$ such that $p' \cdot a$ is almost equivalent to p. To keep $\mathcal{B}$ deterministic, we take the minimal such p' according to an arbitrarily chosen order on the states of $\mathcal{A}$. If such a state p' does not exists, $\mathcal{B}$ moves to $\perp$.

Consider an input tree T. For each prefix w of $\langle T \rangle$, let $\widehat{w}$ be the word obtained from w by successively erasing all two-letter subwords of the form $a\bar{a}$ for $a \in \Gamma$. If w ends with the opening tag of a node x in T, then $\widehat{w}$ is the sequence of labels on the shortest path from the root of T to x. If w ends with the closing tag of a node x in T, then $\widehat{w}$ is the sequence of labels on the shortest path from the root of T to the parent of x (if x is the root of T, then the path is empty). We claim that for every proper nonempty prefix w of $\langle T \rangle$, the state p_w of $\mathcal{B}$ after reading w is a an internal state of $\mathcal{A}$ that is almost equivalent to the state $q_{\widehat{w}}$ of $\mathcal{A}$ after reading $\widehat{w}$, and if the last letter of w is an opening tag, then $p_w = q_{\widehat{w}}$. The claim immediately implies that $\mathcal{B}$ realizes QL, because the first and the last state of $\mathcal{B}$ in the run on $\langle T \rangle$ does not matter.

We prove the claim by induction on $|w|$. The automaton $\mathcal{B}$ begins the computation in the initial state of $\mathcal{A}$. The first letter of $\langle T \rangle$ is some opening tag a. Because $\widehat{a} = a$, we have $p_a = q_{\widehat{a}}$ and p_a is clearly internal. Suppose now that the claim holds for w. If the next letter after w is an opening tag c, applying Lemma 3.3 to the almost equivalent states p_w and $q_{\widehat{w}}$ of $\mathcal{A}$, we get $p_{wc} = p_w \cdot c = q_{\widehat{w}} \cdot c = q_{\widehat{wc}}$, and we are done because $q_{\widehat{w}} \cdot c$ is clearly internal. Suppose that the next letter read by $\mathcal{B}$ is a closing tag $\bar{c}$. We need to prove that there exists an internal state p' in $\mathcal{A}$ such that $p' \cdot c$ is almost equivalent to p_w, and that every such p' is almost equivalent to $q_{\widehat{w\bar{c}}}$. Consider $p' = q_{\widehat{w\bar{c}}}$. Because $w\bar{c}$ is a proper prefix of $\langle T \rangle$, the word $\widehat{w\bar{c}}$ is nonempty; hence, $q_{\widehat{w\bar{c}}}$ is a internal state of $\mathcal{A}$. We also have $q_{\widehat{w\bar{c}}} \cdot c = q_{\widehat{w}}$, and we have assumed that p_w and $q_{\widehat{w}}$ are almost equivalent; hence, $q_{\widehat{w\bar{c}}} \cdot c$ is almost equivalent to p_w. So, indeed, $q_{\widehat{w\bar{c}}}$ is a correct choice for p'. Let us now take any internal p' with $p' \cdot c$ almost equivalent to p_w, and prove that p' is almost equivalent to $q_{\widehat{w\bar{c}}}$. As p_w and $q_{\widehat{w}}$ are almost equivalent by the induction hypothesis, it follows that so are $p' \cdot c$ and $q_{\widehat{w}}$. By Lemma 3.3, $p' \cdot c \cdot b = q_{\widehat{w}} \cdot b = q_{\widehat{w\bar{c}}} \cdot c \cdot b$ for each $b \in \Gamma$. Hence, p' and $q_{\widehat{w\bar{c}}}$ meet. We have already argued that $q_{\widehat{w\bar{c}}}$ is internal, and p' is internal by assumption. Because $\mathcal{A}$ is almost-reversible, we conclude that p' is almost equivalent to $q_{\widehat{w\bar{c}}}$. □

3.2 Hierarchical almost-reversibility

We have already developed intuitions on evaluating RPQs over markup encodings using finite automata. Can we do more using the depth information and the (limited) ability to process it offered by the registers? Using one register and an additional component in the state, we can store the configuration of the simulated automaton in one node on the path from the root to the current node: we store the depth of this node in the register and the state of the simulated automaton in the additional component of the state of

the simulating automaton. When the simulation climbs up to this depth again, we know to which state the simulated automaton should be reverted, regardless of the reversibility assumptions.

Using this feature we can simulate automata whose strongly connected components (SCCs) are singletons (Fig. 3b). Recall that an SCC is a maximal subset X of the state-space such that every state in X is reachable from every other state in X. If each SCC is a singleton, then a run may loop in some states it visits, but it never revisits a state it has left. Hence, in each run there is a bounded number of state changes. The simulating automaton can then represent the whole run of the simulated automaton over the path from the root to the current node by means of the list of state changes and depths at which these changes occurred. Automata with only singleton SCCs capture exactly the class of $\mathcal{R}$*-trivial* languages, named after one of Green's relations from algebraic formal language theory [21]; the intensively studied *piecewise testable languages* [19] form a prominent subclass of $\mathcal{R}$-trivial languages.

As we shall see, the potential of register automata is exhausted by the combination of the above simulation method with the full power of finite automata to simulate a run inside a single SCC. The class of automata that can be simulated this way is captured by the following definition.

Definition 3.6 (Hierarchical almost-reversibility). A deterministic automaton is *hierarchically almost-reversible*, abbreviated as HAR, if every two states from the same SCC that meet inside this SCC are almost equivalent. A regular language is HAR if its minimal automaton is HAR.

By design, HAR languages include all almost-reversible languages (Fig. 3a), and all $\mathcal{R}$-trivial languages (Fig. 3b), but also the language in Fig. 3c which is neither almost-reversible nor $\mathcal{R}$-trivial. The language in Fig. 3d, is not HAR.

As Definition 3.6 is invariant under the complementation of the automaton, we obtain the following.

Lemma 3.7. *The complement of a* HAR *language is* HAR.

Let us see that HAR languages can indeed be handled by depth-register automata.

Lemma 3.8. *If L is a* HAR *language, then QL is a stackless query.*

Proof. Let L be a HAR language and $\mathcal{A}$ its minimal automaton. Like before, we construct a depth-register automaton $\mathcal{B}$ that evaluates QL by maintaining a simulation of the run of $\mathcal{A}$ on the word $\widehat{w}$ labelling the path π from the root to the current node. It applies the method used for $\mathcal{R}$-trivial languages to keep track of the changes of SCCs of $\mathcal{A}$ during the simulated run, and an adaptation of the method for almost-reversible languages to deal with the segments of the simulated run within a single SCC. After processing a prefix w of the encoding of the input tree, for each SCC X of $\mathcal{A}$ visited during the run on $\widehat{w}$, except the current one, the automaton $\mathcal{B}$ stores

- the depth of the deepest node on the path π whose label was read in a state from X during the run on $\widehat{w}$; and
- some state from X that meets in X with the last state from X visited by $\mathcal{A}$ in the run on $\widehat{w}$.

Additionally, if q is the current state of $\mathcal{A}$ after processing $\widehat{w}$ and Y is the SCC of $\mathcal{A}$ that contains q, the automaton $\mathcal{B}$ stores some state $p \in Y$ that meets with q in Y, and $p = q$ after reading each opening tag. Initially, p is the initial state i of $\mathcal{A}$, and nothing else is stored.

Suppose that $\mathcal{B}$ reads an opening tag a and the current depth is d. Because $\mathcal{A}$ is HAR, the states p and q mentioned above are almost equivalent. As $\mathcal{A}$ is minimal, it follows from Lemma 3.3 that $p \cdot a = q \cdot a$. Consequently, $p \cdot a$ is the next state of $\mathcal{A}$. If $p \cdot a \in Y$, we just replace p with $p \cdot a$ and proceed to the next tag. If $p \cdot a$ belongs to some SCC $Z \neq Y$, we also add Y to the list of remembered SCCs, with depth d (loaded to some unused register) and state p, and continue with Z as the current SCC.

Suppose now that $\mathcal{B}$ reads a closing tag $\bar{a}$ and the current depth d is greater than or equal to the maximal recorded depth d'. This indicates that the previous state of $\mathcal{A}$ also belongs to Y. We should now revert $\mathcal{A}$ to some state $q' \in Y$ such that $q' \cdot a = q$, but we do not know which one. Even worse, we do not have access to q, but only to some state $p \in Y$ that meets with q in Y. Nevertheless, we can maintain the invariant by picking any state $p' \in Y$ such that $p' \cdot a \in Y$ is almost equivalent to p. Note first that such states p' exist because q' is one of them: $q' \cdot a = q$ and from the previous case we know that q and p are almost equivalent. To keep $\mathcal{B}$ deterministic we pick the minimal such p' according to some arbitrarily fixed order on the states of $\mathcal{A}$. To prove that every p' is suitable it suffices to show that p' meets with q' in Y. We know that $p \cdot u = q \cdot u \in Y$ for some word u. Because $p' \cdot a$ is almost equivalent to p and $\mathcal{A}$ is minimal, we get $p' \cdot a \cdot u = p \cdot u = q \cdot u = q' \cdot a \cdot u$, and we are done. Hence, $\mathcal{B}$ can replace p with p' and proceed to the next tag.

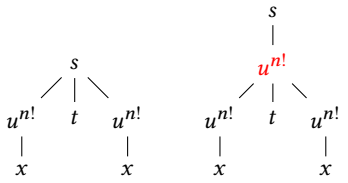
Finally, suppose $\mathcal{B}$ reads a closing tag $\bar{a}$ and the current depth is strictly smaller than the greatest recorded depth d'. This indicates that the previous state of $\mathcal{A}$ belongs to the SCC $X \neq Y$, associated with depth d'. The automaton $\mathcal{A}$ should be reverted to the last state q' from X visited during the run. The simulation does not have access to q', but it has the state p' recorded for X, and we know that p' meets with q' in X. This is sufficient to maintain the invariant: the automaton $\mathcal{B}$ simply replaces p with p', removes X from the list of remembered SCCs marking the register storing the associated depth d' as unused, and proceeds to the next tag with X as the current SCC. □

3.3 Flatness

Not all finite languages are almost-reversible, as witnessed by the one in Fig. 3b. Nevertheless, if L is finite, then AL is registerless. Indeed, a finite automaton can simply simulate the stack up to the depth bounded by the length of the longest word in L. If an opening tag is read when the stack is at its maximum depth, the automaton moves to an all-rejecting sink state. Symmetrically, if L is co-finite (that is, L^c is finite), then EL is registerless. This motivates the following dual notions.

Definition 3.9 (E-flatness and A-flatness). We call a state q *acceptive* (resp. *rejective*) if $q \cdot w$ is accepting (resp. rejecting) for some $w \in \Gamma^*$. A deterministic automaton is *E-flat* (resp. *A-flat*) if for every internal state p and every rejective (resp. acceptive) state q, if p meets with q in q, then p is almost equivalent to q. A *E-flat* (resp. *A-flat*) *language* is a regular language whose minimal automaton is E-flat (resp. A-flat).

Checking that all finite languages (including the one in Fig. 3b) are A-flat, and all co-finite ones are E-flat is an easy exercise. The



(a) Tree S. **(b) Tree S'.**

Figure 4: Fooling trees in Lemma 3.12.

following lemma, connecting flatness to almost-reversibility is not hard either (see Appendix D).

Lemma 3.10. *Let $L \subseteq \Gamma^*$ be a regular language.*

(1) L is A-flat iff L^c is E-flat.

(2) L almost-reversible iff it is both A-flat and E-flat.

More effort is needed to show that E-flatness of L is sufficient to simulate its minimal automaton faithfully enough to support recognizing EL.

Lemma 3.11. *If L is an E-flat language, then EL is a registerless tree language.*

Like for almost-reversible automata, the high-level idea is to maintain the state of the simulated automaton up to almost equivalence, except that we should immediately accept if this state becomes non-rejective. Because the internal structure of E-flat automata is much richer, we additionally need to keep track of transitions that moved the simulated run from one SCC to the next one. Taking transitions backwards when processing closing tags introduces certain ambiguity: the origins of the stored transitions are not single states, but pairs of states guaranteed to be almost equivalent. Full details can be found in Appendix E.

3.4 Inexpressibility

The results established in this section are proved by pumping simultaneously at the level of trees and their encodings, which resembles pumping arguments for context free grammars. To simplify factorizing encodings of trees, for a word $w = a_1 a_2 \cdots a_n \in \Gamma^*$ we let $\bar{w} = \bar{a}_n \cdots \bar{a}_2 \bar{a}_1$ (note the reversed order). Consider the tree S shown in Fig. 4a, keeping in mind that s, t, u, x are words rather than single letters: each node labelled with a word w represents a chain of $|w|$ nodes whose labels form the word w. Then,

$$\langle S \rangle = s u^{n!} x \bar{x} \bar{u}^{n!} t \bar{t} u^{n!} x \bar{x} \bar{u}^{n!} \bar{s}\,.$$

We use S in the proof of the following lemma.

Lemma 3.12. *For a regular language L, if EL is a registerless tree language, then L is E-flat.*

Proof. Suppose that the minimal automaton $\mathcal{A}$ of $L \subseteq \Gamma^*$ is not E-flat. Let i be the initial state of $\mathcal{A}$. Then, there exist words $s, t, u \in \Gamma^+$, $x \in \Gamma^*$ and states p, q such that $i \cdot s = p$, $p \cdot u = q \cdot u = q$, $q \cdot x$ is rejecting, and $p \cdot t$ is accepting iff $q \cdot t$ is rejecting. It follows that for each $k > 0$, $s u^k x \in L^c$, and $st \in L$ iff $s u^k t \in L^c$.

Consider a deterministic finite automaton $\mathcal{B}$ over $\Gamma \cup \bar{\Gamma}$ with n states. It is well known that $r \cdot w^{n!} = r \cdot w^{2 \cdot n!}$ for each nonempty word w and each state r of $\mathcal{B}$ (this is also implied by Lemma 3.15 established later in this section).

Consider the trees S and S' shown in Fig. 4. By the discussion above, exactly one of those trees belongs to EL. Consider the runs of $\mathcal{B}$ on $\langle S \rangle$ and $\langle S' \rangle$. Suppose that on $\langle S \rangle$ we have

$$q_0 \xrightarrow{s u^{n!}} q_1 \xrightarrow{x \bar{x} \cdot \bar{u}^{n!} \cdot t \bar{t} \cdot u^{n!} x \bar{x} \bar{u}^{n!}} q_2 \xrightarrow{\bar{s}} q_3\,.$$

Then, by the choice of n, we have

$$q_0 \xrightarrow{s u^{n!}} q_1 \xrightarrow{u^{n!}} q_1 \xrightarrow{x \bar{x} \cdot \bar{u}^{n!} \cdot t \bar{t} \cdot u^{n!} x \bar{x} \bar{u}^{n!}} q_2 \xrightarrow{\bar{u}^{n!}} q_2 \xrightarrow{\bar{s}} q_3\,.$$

It follows that $\mathcal{B}$ accepts $\langle S \rangle$ iff it accepts $\langle S' \rangle$. Consequently, $\mathcal{B}$ does not recognize EL. □

The missing implication in Theorem 3.1 is also proved by pumping, but requires considerably more effort because this time we need to fool a depth-register automaton. Before we dive into it, we prepare some simple tools helping to analyze runs of such automata; proofs of the auxiliary lemmas can be found in Appendix F.

For configurations $c = (q, d, \eta)$ and $c' = (q', d', \eta')$ of a depth-register automaton $\mathcal{B}$ we write $c \sim c'$ if $q = q'$.

For $-\infty \le i \le j \le \infty$, we write $c \approx_{i,j} c'$ if $c \sim c'$ and for each register ξ one of the following conditions holds:

- $\eta'(\xi) - d' = \eta(\xi) - d$;
- $\eta(\xi) - d < i$ and $\eta'(\xi) - d' < i$ and $\eta(\xi) = \eta'(\xi)$;
- $\eta(\xi) - d > j$ and $\eta'(\xi) - d' > j$.

We let $\|\varepsilon\| = 0$ and inductively $\|wa\| = \|w\| + 1$ and $\|w\bar{a}\| = \|w\| - 1$ for all $a \in \Gamma$ and $w \in (\Gamma \cup \bar{\Gamma})^*$. For nonempty w we also define

$$\lfloor w \rfloor = \min_{\varepsilon \neq u \le w} \|u\|\,, \qquad \lceil w \rceil = \max_{\varepsilon \neq u \le w} \|u\|\,,$$

where $u \le w$ means that u is a prefix of w. Note that for all w,

$$\lfloor w \rfloor \le \|w\| \le \lceil w \rceil\,.$$

Lemma 3.13. *Suppose that $c_1 \approx_{i,j} c_2$. For every word w such that $i \le \lfloor w \rfloor \le \lceil w \rceil \le j$, it holds that $c_1 \cdot w \approx_{i - \|w\|, j - \|w\|} c_2 \cdot w$.*

A word $x \in (\Gamma \cup \bar{\Gamma})^+$ is *descending* if $1 = \lfloor x \rfloor \le \lceil x \rceil = \|x\|$ and it is *ascending* if $-1 = \lceil x \rceil \ge \lfloor x \rfloor = \|x\|$. Descending words generalize words from Γ^+, and ascending words generalize words from $\bar{\Gamma}^+$. For $i, j \in \mathbb{Z}_\infty = \mathbb{Z} \cup \{-\infty, \infty\}$ we let

$$[i, j] = \{k \in \mathbb{Z}_\infty \mid i \le k \le j\}\,, \quad (i, j] = \{k \in \mathbb{Z}_\infty \mid i < k \le j\}\,,$$

and analogously for $[i, j)$ and (i, j).

Lemma 3.14. *Let $c_i = (q_i, d_i, \eta_i)$ with $i \in [1; 4]$ be configurations of a depth-register automaton $\mathcal{B}$ and let $y, z \in (\Gamma \cup \bar{\Gamma})^+$ be descending words such that $c_1 \xrightarrow{y} c_2 \xrightarrow{z} c_3 \xrightarrow{y} c_4$. If $\mathrm{img}(\eta_1) \subseteq (-\infty; d_1]$ and $c_1 \sim c_3$, then $\mathrm{img}(\eta_4) \cap (d_1; d_2] = \emptyset$.*

Lemma 3.15. *Let $\mathcal{B}$ be a depth-register automaton with k states and ℓ registers, and let $n \ge k \cdot (\ell + 1)$. For every configuration $c = (q, d, \eta)$ of the automaton $\mathcal{B}$ and every descending or ascending word $x \in (\Gamma \cup \bar{\Gamma})^+$, if*

$$\mathrm{img}(\eta) \cap \left[d + \left\lfloor x^{3 \cdot n!} \right\rfloor ; d + \left\lceil x^{3 \cdot n!} \right\rceil \right] = \emptyset\,,$$

then

(1) $c \cdot x^{n!} \sim c \cdot x^{n!} \cdot x^{n!}$; and

(2) $c \cdot x^{n!} \cdot x^{n!} \approx_{\lfloor x^{n!} \rfloor - \|x^{n!}\|, \lceil x^{n!} \rceil - \|x^{n!}\|} c \cdot x^{n!} \cdot x^{n!} \cdot x^{n!}$.

Lemma 3.16. *For each regular language L, if EL is a stackless tree language, then L is* HAR.

Proof. Again, we prove the contrapositive. Suppose $L \subseteq \Gamma^*$ is not HAR. Then, its minimal automaton $\mathcal{A}$ admits states p, q, and r in the same SCC Y such that for some word u and some non-empty word t, we have $r = p \cdot u = q \cdot u$ and $p \cdot t$ is accepting and $q \cdot t$ is non-accepting (in particular, $p \neq q$). Then, there exist v and w such that $r \cdot v = p$ and $r \cdot w = q$. Finally, by minimality, all states are reachable from the initial state, so there exists a word s such that $i \cdot s = r$. Because Y contains two different states, it is a non-trivial SCC. Consequently, for each state $p' \in Y$ there exists a nonempty looping word; that is, a word $w' \neq \varepsilon$ such that $p' \cdot w' = p'$. By appending suitable looping words if necessary, we can assume that the words s, u, v, w are nonempty as well. Additionally, it will be convenient to assume that $|u| \geq |t|$; this can be ensured by appending $|t|$ copies of the appropriate looping word to u. The resulting fragment of the automaton $\mathcal{A}$ is shown in the top left corner of Fig. 5. We have

$$s(wu + vu)^* vt \subseteq L\,, \qquad s(wu + vu)^* wt \subseteq L^{\mathrm{c}}\,.$$

Consider a depth-register automaton $\mathcal{B}$ over $\Gamma \cup \overline{\Gamma}$ with k states and ℓ registers. Let $n = k \cdot (\ell + 1)$. We shall construct a fooling pair of trees by unravelling the fooling gadget. The trees are shown in Fig. 5. The original tree R, is build from: (i) a tree R_0 consisting of a single branch labelled by the word s, (ii) trees $R_1, \ldots, R_{2 \cdot n!+1}$ that are isomorphic copies of the same tree, and (iii) a tree $R_{2 \cdot n!+2}$ consisting of a single branch labelled by the word wt. Each branch of R is labelled by a word from $s(wu + vu)^* wt \subseteq L^{\mathrm{c}}$, which means that $R \notin EL$. The pumped tree R' is obtained by inserting an additional segment labelled by $(uv)^{n!}$ in $R_{n!+1}$, just before the branching; we will write $R'_{n!+1}$ for thus modified $R_{n!+1}$. The modification introduces a branch labelled by a word from $s(wu + vu)^* vt \subseteq L$, which means that $R' \in EL$. We will show that the automaton $\mathcal{B}$ cannot distinguish $\langle R \rangle$ from $\langle R' \rangle$, by analyzing the respective runs in parallel. The crucial moments of the analysis will be configurations $c_i = (q_i, d_i, \eta_i)$, $c'_i = (q'_i, d'_i, \eta'_i)$, and $c''_i = (q''_i, d''_i, \eta''_i)$, depicted (with the exception of c_6) in brown in Fig. 5: configurations to the left of edges are visited when going down and those to the right when going up.

Let x be the prefix of $\langle R_1 \rangle$ ending at the opening tag of the rightmost leaf of R_1. Because $|t| \leq |u|$, the rightmost branch of R_1 is at least as long as both other branches, which implies that x is descending. Clearly, so is $y = wu(vu)^{2 \cdot n!} \in \Gamma^+$. Consider the following initial segments of the runs of $\mathcal{B}$ over $\langle R \rangle$ and $\langle R' \rangle$:

$$c_0 \xrightarrow{sx^{n!}} c_1 \xrightarrow{y^{n!}} c_2 \xrightarrow{w} c_3 \xrightarrow{(uv)^{2 \cdot n!}} c_4 \xrightarrow{u} c_5 \xrightarrow{y^{n!-1}} c_6 \xrightarrow{y} c_7\,,$$
$$c_0 \xrightarrow{sx^{n!}} c_1 \xrightarrow{y^{n!}} c_2 \xrightarrow{w} c_3 \xrightarrow{(uv)^{3 \cdot n!}} c'_4 \xrightarrow{u} c'_5 \xrightarrow{y^{n!-1}} c_6 \xrightarrow{y} c'_7\,.$$

Let $\delta = |(uv)^{n!}|$. As all words over the arrows are descending, we have

$$\mathrm{img}(\eta_i) \subseteq [-\infty; d_i]\,, \quad \mathrm{img}(\eta'_j) \subseteq [-\infty; d'_j]\,, \quad d'_j = d_j + \delta \qquad (1)$$

for all $i \in [0; 7]$ and $j \in [4; 7]$. Condition (1) allows us to apply Lemma 3.15 to configuration c_3 and the descending word uv, and conclude that $c_4 \approx_{1-\delta,0} c'_4$. By (1), this can be strengthened to

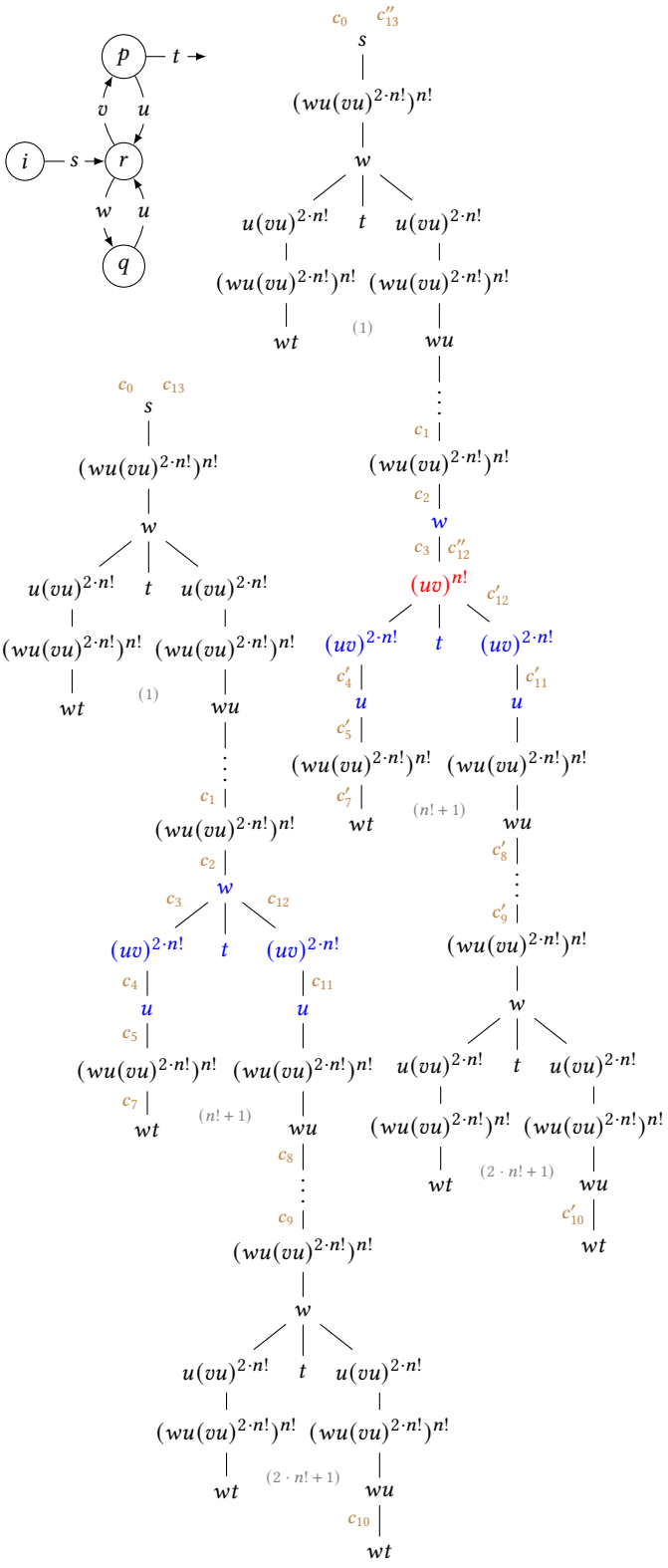


Figure 5: Non-HAR gadget and fooling trees in Lemma 3.16.

$c_4 \approx_{1-\delta,\infty} c'_4$. By Lemma 3.13, we get

$$c_7 \approx_{1-\|(uv)^{n!}u\cdot y^{n!}\|,\infty} c'_7 . \tag{2}$$

Applying Lemma 3.15 to c_2 and y, we get $c_2 \sim c_6$. Hence, we can apply Lemma 3.14 to configurations c_2, c_5, c_6, c_7 and descending words y and $y^{n!-1}$. Combining the result with (2), we get

$$\mathrm{img}(\eta_7) \cap \left(d_2; d_5\right] = \emptyset , \qquad \mathrm{img}(\eta'_7) \cap \left(d_2; d'_5\right] = \emptyset . \tag{3}$$

Consequently, from (2) we can also conclude

$$c_7 \approx_{1-\|y^{n!+1}\|,\infty} c_7 . \tag{4}$$

Let $y' = wu(vu)^{3\cdot n!}$ and take x_0 such that $y^{2\cdot n!+1}\cdot x_0 = x$. Consider

$$c_0 \xrightarrow{sx^{n!}} c_1 \xrightarrow{y^{n!}\cdot y\cdot y^{n!}} c_7 \xrightarrow{x_0} c_8 \xrightarrow{x^{n!-1}} c_9 \xrightarrow{x} c_{10} ,$$

$$c_0 \xrightarrow{sx^{n!}} c_1 \xrightarrow{y^{n!}\cdot y'\cdot y^{n!}} c'_7 \xrightarrow{x_0} c'_8 \xrightarrow{x^{n!-1}} c'_9 \xrightarrow{x} c'_{10} .$$

Note that condition (1) holds for all $i, j \in [8; 10]$. From (4) via Lemma 3.13 we get

$$c_{10} \approx_{1-\|y^{n!+1}wux^{n!}\|,\infty} c'_{10} . \tag{5}$$

Applying Lemma 3.15 to configuration $c_0 \cdot s$ and the descending word x, we get $c_1 \sim c_9$. Applying Lemma 3.14 to configurations c_1, c_8, c_9, c_{10} and the descending words x and $x^{n!}$, and combining the result with (5), we get

$$\mathrm{img}(\eta_{10}) \cap (d_1; d_8] = \emptyset , \qquad \mathrm{img}(\eta'_{10}) \cap (d_1; d'_8] = \emptyset . \tag{6}$$

Hence, we can strengthen (5) to

$$c_{10} \approx_{1-\|x^{n!+1}\|,\infty} c'_{10} . \tag{7}$$

Let $\bar{x} = \bar{u}\bar{w}\bar{y}^{2\cdot n!+1}$; that is, $x\bar{x} = \langle R_1 \rangle$. Consider

$$c_{10} \xrightarrow{wt\bar{t}\,\bar{w}\cdot\bar{x}^{n!}\cdot\bar{u}\bar{w}\bar{y}^{n!}\cdot\bar{u}} c_{11} \xrightarrow{(\bar{v}\bar{u})^{2\cdot n!}} c_{12} \xrightarrow{\bar{w}\cdot\bar{y}^{n!}\cdot\bar{x}^{n!}\cdot\bar{s}} c_{13} ,$$

$$c'_{10} \xrightarrow{wt\bar{t}\,\bar{w}\cdot\bar{x}^{n!}\cdot\bar{u}\bar{w}\bar{y}^{n!}\cdot\bar{u}} c'_{11} \xrightarrow{(\bar{v}\bar{u})^{2\cdot n!}} c'_{12} \xrightarrow{(\bar{v}\bar{u})^{n!}} c''_{12} \xrightarrow{\bar{w}\cdot\bar{y}^{n!}\cdot\bar{x}^{n!}\cdot\bar{s}} c''_{13} .$$

We have $d'_i = d_i + \delta$ for $i \in [10; 12]$ and $d''_i = d_i$ for $i \in [12; 13]$. By Lemma 3.13, we have $c_{12} \approx_{1-\|w\|,\infty} c'_{12}$. As from (6) it follows that $\mathrm{img}(\eta_{12}) \cap (d_1; d_{12}) = \mathrm{img}(\eta'_{12}) \cap (d_1; d'_{12}) = \emptyset$, we also have

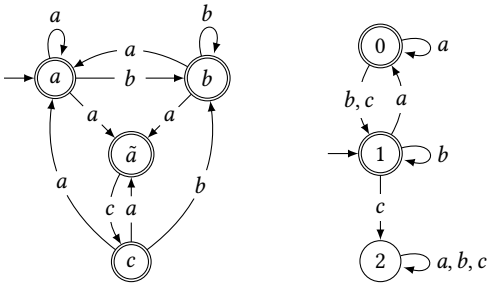
$$c_{12} \approx_{0,\infty} c'_{12} . \tag{8}$$

Applying Lemma 3.15 to configuration c'_{11} and the ascending word $\bar{v}\bar{u}$, we get $c'_{12} \approx_{0,\delta-1} c''_{12}$. In combination with (8) this implies $c_{12} \approx_{0,\delta-1} c''_{12}$. Because $d_{12} = d''_{12}$, it follows that $c_{12} \approx_{-\infty,\delta-1} c''_{12}$. By Lemma 3.13, this implies $c_{13} \sim c''_{13}$. □

4 DISCUSSION

4.1 Tree languages defined by DTDs

Our characterization results shed some light on the registerlessness of DTDs, studied in [25] (called *recognizability* there). A *DTD D over* Γ consists of an initial symbol $a_0 \in \Gamma$ and, for each $a \in \Gamma$, a production of the form $a \to L_a$ where L_a is a regular language over Γ (typically represented as a regular expression). It defines the set of trees T over Γ that have a_0 in the root and for each a-labelled node v in T, the labels of v's children read from left to right form a word in L_a. A *specialized DTD over* Γ consists of a DTD D' over Γ' and an alphabet projection $\pi : \Gamma' \to \Gamma$; the language it defines is the projection of the language defined by D' to the alphabet Γ.



(a) Original automaton. **(b) Minimal automaton.**

Figure 6: Automata corresponding to the specialized DTD $a \to (a + b + \tilde{a})^*$, $b \to (a + b + \tilde{a})^*$, $\tilde{a} \to c^*$, $c \to (a + b)^*$ with alphabet projection $a \mapsto a$, $\tilde{a} \mapsto a$, $b \mapsto b$, $c \mapsto c$.

Languages of the form AL capture (resp. capture precisely) all tree languages definable by DTDs (resp. specialized DTDs) using only productions of the forms

$$a \to (b_1 + \cdots + b_n)^* , \quad a \to (b_1 + \cdots + b_n)^+ .$$

This is a severely restricted, yet non-trivial and practically relevant, special case of the setting considered in [25]. Let us refer to such DTDs as path DTDs. A path DTD is almost an automaton recognizing allowed paths: use (specialized) symbols as states, add a transition from a to each b_i over symbol b_i (or its projection $\pi(b_i)$ in the case of specialized DTDs), and let a be accepting if the production uses $*$, and non-accepting if it uses $+$ (see Fig. 6a).

It can be shown that under restriction to path DTDs, the first necessary condition for registerlessness proposed in [25] reduces to the assumption that the corresponding automaton is HAR, and the second one amounts to A-flatness. Segoufin and Vianu show that the first necessary condition is also sufficient under the restriction to fully-recursive DTDs, which correspond to automata that have only two non-trivial SCCs: one contains the initial state, and the other is an all-rejecting sink. For such automata, HAR is equivalent to A-flat, which makes their result a special case of Theorem 3.2 (2) (in the limited special case of path DTDs). Segoufin and Vianu also conjecture, that the two necessary conditions together are sufficient for all DTDs. Theorem 3.2 (2) confirms this conjecture in the special case of path DTDs. Let us remark that A-flatness works also for languages defined by specialized path DTDs, but the corresponding automaton must be determinized and minimized before the criterion is applied, as witnessed by the specialized DTD in Fig. 6 which gives an A-flat non-deterministic automaton, which is not A-flat any more after determinizing and minimizing.

4.2 A different encoding of trees

An alternative way to serialize tree-structured data, used for instance in JSON, is the *term encoding*, in which the information about the label is included only in opening tags. For instance, instead of $aba\bar{a}a\bar{a}\bar{b}c\bar{c}\bar{a}$ we would have `a{b{a{}a{}}c{}}`, where `a{`, `b{`, `c{` are opening tags, and `}` is the universal closing tag. Streaming processing under this encoding is harder, but analyzing it is easier. An effective characterization of regular tree languages that are registerless under the term encoding is given in [1].

Our treatment can be easily adapted to the term encoding by adjusting the definition of when two states meet: we say that states p and q *blindly meet* in state r if there exist words $u_1, u_2 \in \Gamma^*$ such that $|u_1| = |u_2|$ and $p \cdot u_1 = q \cdot u_2 = r$. By replacing 'meet' with 'blindly meet' in Definitions 3.4, 3.6 and 3.9, we get the definitions of the syntactic classes of *blindly almost-reversible*, *blindly* HAR, *blindly A-flat*, and *blindly E-flat* word languages. Theorems 3.1 and 3.2 then hold for the term encoding with all syntactic classes of word languages replaced by their blind analogues (see Appendix G). Based on this, it can be checked by direct examination of the automata in Fig. 3 that also under the term encoding, the first RPQ from Example 2.12 is registerless, the following two are stackless but not registerless, and the last one is not stackless. Nevertheless, 'blind' classes are much more restricted than their originals: all $\mathcal{R}$-trivial languages are blindly HAR, but the possibilities of backtracking inside an SCC are very limited. For example, the minimal automaton shown in Fig. 2 is reversible, but not blindly-HAR; this means that the language $(b^*ab^*ab^*)^*$ this automaton recognizes is registerless under the markup encoding, but not even stackless under the term encoding. This is the cost of succinctness.

4.3 Outlook

In this work we have proposed an intermediate model for processing streamed trees, increasing the expressive power of finite automata considerably while sparing us the maintenance of the stack. We have effectively characterized unary RPQs that can be realized in this model, and those that can be realized by finite automata. The latter leads to a partial solution of the weak validation problem posed by Segoufin and Vianu [25].

The weak validation problem remains the most intriguing theoretical challenge in the area. Other salient problems are to characterize (effectively) stackless tree languages among regular ones and, conversely, regular tree languages among stackless ones. The former is more relevant for query and schema processing, but the latter may provide some useful insights as well.

Solving these problems for all regular tree languages might be very hard, but for more restricted, yet practically relevant, subclasses it might be easier. For instance, it would be very useful to be able to decide if a given XPath expression is stackless or registerless (both as a boolean query and as a unary query). Examples 2.7, 2.9 and 2.10 suggest that stackless XPath expressions would have to use child, next-sibling, following-sibling, and negation extremely cautiously, but this might be alleviated by including schema information into the setting.

Applying our results on the term encoding to JSON would also involve incorporating rudimentary schema information, as in JSON siblings either have different labels, or have no labels at all.

Finally, a major question is how to vectorize. A first step would be to uncover the local parallelism of pushdown and depth-register automata, in the spirit of [15]. Closing the distance to actual applications will require replacing circuits with a more faithful abstraction of the capabilities of CPUs.

ACKNOWLEDGMENTS

This work was supported by Poland's National Science Centre grant 2018/30/E/ST6/00042.

REFERENCES

[1] Vince Bárány, Christof Löding, and Olivier Serre. Regularity problems for visibly pushdown languages. In *Proc. STACS 2006*, pages 420–431. Springer, 2006.
[2] David A. Mix Barrington and James C. Corbett. On the relative complexity of some languages in NC^1. *Inf. Process. Lett.*, 32(5):251–256, 1989.
[3] Angela Bonifati, George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets. *Querying Graphs.* Morgan & Claypool Publishers, 2018.
[4] Robert D. Cameron, Ehsan Amiri, Kenneth S. Herdy, Dan Lin, Thomas C. Shermer, and Fred Popowich. Parallel scanning with bitstream addition: An XML case study. In *Proc. Euro-Par 2011*, pages 2–13. Springer, 2011.
[5] Cristiana Chitic and Daniela Rosu. On validation of XML streams using finite state machines. In *Proc. WebDB 2004*, pages 85–90. ACM, 2004.
[6] Denis Debarbieux, Olivier Gauwin, Joachim Niehren, Tom Sebastian, and Mohamed Zergaoui. Early nested word automata for XPath query answering on XML streams. *Theor. Comput. Sci.*, 578:100–125, 2015.
[7] Patrick Dymond. Input-driven Languages Are in Log N Depth. *Inf. Process. Lett.*, 26(5):247–250, January 1988.
[8] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj D. Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. Anatomy of high-performance deep learning convolutions on SIMD architectures. In *Proc. SC 2018*, pages 66:1–66:12. IEEE / ACM, 2018.
[9] Alejandro Grez, Cristian Riveros, and Martín Ugarte. A formal framework for complex event processing. In *Proc. ICDT 2019*, pages 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
[10] Sascha Grunert and Daniel Schmidt. A comparison of regex engines, 2017. https://rust-leipzig.github.io/regex/2017/03/28/comparison-of-regex-engines/.
[11] Ashish Kumar Gupta and Dan Suciu. Stream processing of XPath queries with predicates. In *Proc. SIGMOD 2003*, pages 419–430. ACM, 2003.
[12] Yeye He, Siddharth Barman, and Jeffrey F. Naughton. On load shedding in complex event processing. In *Proc. ICDT 2014*, pages 213–224. OpenProceedings.org, 2014.
[13] Eryk Kopczynski. Invisible pushdown languages. In *Proc. LICS 2016*, pages 867–872. ACM, 2016.
[14] Geoff Langdale and Daniel Lemire. Parsing gigabytes of JSON per second. *VLDB J.*, 28(6):941–960, 2019.
[15] Filip Murlak, Charles Paperman, and Michal Pilipczuk. Schema validation via streaming circuits. In *Proc. PODS 2016*, pages 237–249. ACM, 2016.
[16] Damian Niwinski and Igor Walukiewicz. A gap property of deterministic tree languages. *Theor. Comput. Sci.*, 303(1):215–231, 2003.
[17] Dan Olteanu. SPEX: streamed and progressive evaluation of XPath. *IEEE Trans. Knowl. Data Eng.*, 19(7):934–949, 2007.
[18] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Filter before you parse: Faster analytics on raw data with sparser. *Proc. VLDB Endow.*, 11(11):1576–1589, 2018.
[19] Jean-Eric Pin. Proprietes syntactiques du produit non ambigu. In *Proc. ICALP 1980*, pages 483–499. Springer, 1980.
[20] Jean-Eric Pin. On reversible automata. In *Proc. LATIN 1992*, pages 401–416. Springer, 1992.
[21] Jean Eric Pin and Raymond E. Miller. *Varieties Of Formal Languages.* Plenum Publishing Co., 1986.
[22] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD vectorization for in-memory databases. In *Proc. SIGMOD 2015*, pages 1493–1508. ACM, 2015.
[23] Gang Ren, Peng Wu, and David A. Padua. An empirical study on the vectorization of multimedia applications for multimedia extensions. In *Proc. IPDPS 2005*. IEEE, 2005.
[24] Luc Segoufin and Cristina Sirangelo. Constant-memory validation of streaming XML documents against DTDs. In *Proc. ICDT 2007*, pages 299–313. Springer, 2007.
[25] Luc Segoufin and Victor Vianu. Validating streaming XML documents. In *Proc. PODS 2002*, pages 53–64. ACM, 2002.
[26] Dan Suciu. From searching text to querying XML streams. *J. Discrete Algorithms*, 2(1):17–32, 2004.
[27] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on CPUs, 2011. *Deep Learning and Unsupervised Feature Learning Workshop* @ NIPS 2011.
[28] Burchard von Braunmühl and Rutger Verbeek. Input-driven languages are recognized in log n space. In *Proc. FCT 1983*, pages 40–51. Springer, 1983.
[29] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *Proc. NSDI 2019*, pages 631–648. USENIX Association, 2019.
[30] Haopeng Zhang, Yanlei Diao, and Neil Immerman. On complexity and optimization of expensive queries in complex event processing. In *Proc. SIGMOD 2014*, pages 217–228. ACM, 2014.
[31] Yichun Zhang. Regex engine matching speed benchmark, 2015. http://openresty.org/misc/re/bench/.
[32] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *Proc. SIGMOD 2002*, pages 145–156. ACM, 2002.

A PROOF OF PROPOSITION 2.3

Let us recall the statement.

Proposition A.1. *Restricted depth-register automata recognize regular tree languages.*

Proof. Consider a restricted depth-register automaton

$$\mathcal{A} = (\Gamma, Q, q_{init}, F, \Xi, \delta)\,.$$

The run of $\mathcal{A}$ on a tree T can be represented by means of an auxiliary labelling of the nodes of T with elements of

$$\left(2^{\Xi} \times Q\right) \times 2^{\Xi} \times \left(2^{\Xi} \times Q\right)$$

where for each node v in T, if v gets auxiliary label

$$\big((X, p), Y, (Z, q)\big)$$

then

- upon reading the opening tag of v, $\mathcal{A}$ loads the current depth to registers in X and moves to state p;
- when processing the infix of $\langle T \rangle$ delimited (exclusively) by the opening and closing tags of v, $\mathcal{A}$ loads some current depth to exactly those registers that belong to Y;
- upon reading the closing tag of v, $\mathcal{A}$ loads the current depth to registers in Z and moves to state q.

In what follows, we shall refer to q as the *exit state* of v.

The correctness of the auxiliary labelling can be equivalently expressed in a more local way, relying on the transition function of $\mathcal{A}$. Suppose that a node v has label a in T and auxiliary label $\big((X, p), Y, (Z, q)\big)$, and its children $v_1, v_2, \ldots, v_n$ have labels $a_1, a_2, \ldots, a_n$ in T and auxiliary labels $\big((X_i, p_i), Y_i, (Z_i, q_i)\big)$ for $i \in \{1, 2, \ldots, n\}$. Then,

$$Y = \bigcup_{i=1}^{n} X_i \cup Y_i \cup Z_i$$

and for all $i \in \{1, 2, \ldots, n\}$,

$$(X_i, p_i) = \delta(p_i', a_i, \Xi, \emptyset)\,,$$
$$(Z_i, q_i) = \delta(q_i', a_i, \Xi \setminus (X_i \cup Y_i), X \cup Z_1 \cup \ldots Z_{i-1} \cup X_i \cup Y_i)\,,$$

where

- $p_1' = p$ and $p_i' = q_{i-1}$ for $i \in \{2, \ldots, n\}$;
- $q_i' = p_i$ if v_i is a leaf and otherwise q_i' is the exit state of the last child of v_i.

If v is the root of T, it must also hold that

$$(X, p) = \delta(q_{init}, a, \Xi, \emptyset) \quad \text{and} \quad (Z, q) = \delta(q', a, \Xi \setminus Y, \Xi)\,,$$

where $q' = p$ if v is a leaf and otherwise q' is the exist state of the last child of v (that is, $q' = q_n$). The rephrased condition is equivalent to the original one precisely because in a restricted depth-register automaton we have the guarantee that $X_i \cup Y_i \subseteq Z_i$.

To show that the tree language recognized by $\mathcal{A}$ is regular it suffices to observe that it can be recognized by a nondeterministic tree automaton that guess an auxiliary labelling of the input tree, checks its correctness by verifying the rephrased condition, and accepts if the second state in the auxiliary label of the root belongs to F. □

B PROOF OF PROPOSITION 2.11

Let us recall the statement of the proposition.

Proposition B.1. *The class of stackless queries invariant under sibling order is contained in the class of RPQs.*

Proof. Consider a query Q invariant under sibling order, realized by a depth-register automaton $\mathcal{B}$. It follows immediately, that Q is fully described by the answers it gives on the leftmost branch of every tree. But these answers are determined by the word on the path from the root to the current node. Hence, Q is a path query and it is fully described by its behaviour on single-branch trees. Consider the run of $\mathcal{B}$ on the prefix of the encoding of such a tree, consisting of all opening tags. In such a run, the current depth is always strictly greater than all values stored in the registers, so the registers can be eliminated from the automaton. Over single-branch trees, the resulting finite automaton over $\Gamma \cup \bar{\Gamma}$ selects the same nodes as $\mathcal{B}$. By restricting the alphabet to Γ, we obtain an automaton recognizing L. □

C PROOF OF PROPOSITION 2.13

We recall the formulation of the proposition.

Proposition C.1. *It is decidable if the query realized by a given restricted depth-register automaton is an RPQ.*

Proof. By Proposition 2.11, a stackless query is an RPQ iff it is a path query. We phrase the argument for the latter property.

A *marked tree* over Γ is a tree over $\Gamma \times \{0, 1\}$; *marked nodes* in such a tree are those with labels from $\Gamma \times \{1\}$. Let us fix a unary query Q. For a tree T over Γ we let T_Q be the marked tree over Γ obtained from T by marking nodes from $Q(T)$. Let M_Q be the set of all such T_Q with T ranging over all trees over Γ. The query Q is a path query if and only if there exists a language L over $\Gamma \times \{0, 1\}$ such that $M_Q = M_L$, where M_L is the set of marked trees over Γ where each direct path from the root to a marked node is labelled with a word from L. Moreover, if Q is a path query, then we can take for L the language L_Q obtained by restricting M_Q to trees that consist of a single branch with marked leaf. Hence, Q is a path query if and only if $M_Q = M_{L_Q}$.

It is easy to turn this characterization into an algorithm. Suppose that we are given a restricted depth-register automaton $\mathcal{A}$ and let Q be the query it realizes. Based on (the proof of) Proposition 2.3, it is easy to construct a tree automaton $\mathcal{B}$ recognizing M_Q. Next, we intersect $\mathcal{B}$ with a tree automaton recognizing single-branch trees with marked leaf, interpret the result as a word automaton, and thus obtain an automaton C that recognizes the language L_Q. Finally, we easily turn C into a tree automaton $\mathcal{D}$ recognizing M_{L_Q}. Thus, testing if the query realized by $\mathcal{A}$ is a path query reduces to testing if the tree automata $\mathcal{B}$ and $\mathcal{D}$ are equivalent, which is well known to be decidable. □

D PROOF OF LEMMA 3.10

Let us recall the statement of the lemma.

Lemma D.1. *Let $L \subseteq \Gamma^*$ be a regular language.*

(1) L is A-flat iff L^c is E-flat.

(2) L almost-reversible iff it is both A-flat and E-flat.

Proof. Let $\mathcal{A}$ be the minimal automaton of L. Then $\mathcal{A}^{\mathrm{c}}$, obtained from $\mathcal{A}$ by swapping accepting and rejecting states, is the minimal automaton of L^{c}. A state q is acceptive in $\mathcal{A}$ iff it is rejective in $\mathcal{A}^{\mathrm{c}}$. It follows that $\mathcal{A}$ is A-flat iff $\mathcal{A}^{\mathrm{c}}$ is E-flat.

For the second part, observe that states p and q in Definition 3.9 are internal, so every almost-reversible automaton is A-flat and E-flat. For the converse, consider a minimal automaton $\mathcal{A}$ that is A-flat and E-flat. We begin with an auxiliary claim.

We call an SCC X a *sink* if for each $q \in X$ and each $u \in \Gamma^*$, $q \cdot u \in X$. We claim that if a sink SCC X is reachable from an internal state p, then X contains a state q that is almost equivalent to p. Indeed, suppose that $p \cdot w \in X$. Because X is a sink, $p \cdot w^n \in X$ for all $n > 0$. Consequently, there exist $n, k > 0$ such that $p \cdot w^n = p \cdot w^n \cdot w^k$. Moving n positions backwards in the cyclic list of states $p \cdot w^n, p \cdot w^{n+1}, \ldots, p \cdot w^{n+k-1}$, starting from $p \cdot w^n$, we find a state $q = p \cdot w^{n+k-n \bmod k} \in X$ that meets with p. Because X is a sink, p and q can only meet in some $r \in X$. But then p and q also meet in q. Because q is either rejective or acceptive, and $\mathcal{A}$ is both E-flat and A-flat, it follows that p and q are almost equivalent.

To see that $\mathcal{A}$ is almost-reversible, take two internal states p_1 and p_2 that meet. Then, p_1 and p_2 meet in some sink SCC X. Consequently, there exists a non-empty word w and state $r \in X$ such that $p_1 \cdot w = p_2 \cdot w = r$. By the auxiliary claim, X contains states q_1 and q_2 that are almost equivalent to p_1 and p_2, respectively. By Lemma 3.3 and the minimality of $\mathcal{A}$, we get $q_1 \cdot w = p_1 \cdot w = r = p_2 \cdot w = q_2 \cdot w$; that is, the states $q_1, q_2 \in X$ meet in X. Consequently, q_1 meets with q_2 in q_2, and because q_1 is obviously internal, it follows by E-flatness or A-flatness that q_1 and q_2 are almost equivalent. It follows that p_1 and p_2 are almost equivalent, too. □

E PROOF OF LEMMA 3.11

Let us recall the statement of the lemma.

Lemma E.1. *If L is an E-flat language, then EL is a registerless tree language.*

Proof. Let $\mathcal{A}$ be the minimal automaton of L. We first construct an automaton $\mathcal{B}$ simulating $\mathcal{A}$ in a certain precise sense, and then we turn $\mathcal{B}$ into an automaton recognizing EL.

Like in the simulation of almost-reversible automata, the high-level idea is to maintain the state of $\mathcal{A}$ after processing $\widehat{w}$ up to almost equivalence, except that if at any point the maintained state becomes non-rejective, the simulating automaton moves to an all-accepting sink state $\top$. But because the internal structure of E-flat automata is much richer then that of almost-reversible ones, the simulating automaton $\mathcal{B}$ needs more information.

After reading a prefix w of the encoding of the input tree, the simulating automaton $\mathcal{B}$ will store a *synopsis* of the run of $\mathcal{A}$ on $\widehat{w}$. The goal of the synopsis is to list the transitions that moved the run from one SCC of $\mathcal{A}$ to the next one. However, because the automaton $\mathcal{A}$ is not reversible, taking the transitions backwards when processing closing tags will introduce certain ambiguity into the stored transitions. Namely, the origins of the transitions will be *split states*, defined as pairs (p, q) such that q is rejective and either $p = q$ or p is internal and meets with q in q. E-flatness guarantees that for each split state (p, q), the states p and q are almost equivalent. By minimality, transitions from split states have unambiguous targets.

A *split transition* is a tuple (p, q, a, r) such that (p, q) is a split state and $p \cdot a = q \cdot a = r$. A *synopsis* for $\mathcal{A}$ is an alternating sequence of state triples and letters, written as

$$(r_0, p_0, q_0) \xrightarrow{a_1} (r_1, p_1, q_1) \xrightarrow{a_2} \cdots \xrightarrow{a_\ell} (r_\ell, p_\ell, q_\ell), \tag{9}$$

such that r_0 is the initial state of $\mathcal{A}$, each $(p_i, q_i, a_{i+1}, r_{i+1})$ is a split transition in $\mathcal{A}$, (p_ℓ, q_ℓ) is a split state in $\mathcal{A}$, and

- for each $i < \ell$, the states q_i and r_{i+1} are in different SCCs;
- for each $i \leq \ell$, q_i belongs to the SCC of r_i and either p_i belongs to the SCC of r_i or $i > 0$ and $p_i = p_{i-1} = q_{i-1}$.

Observe that the states q_i represent a chain of different SCCs, so $\ell + 1$ is bounded by the depth of the DAG of SCCs of $\mathcal{A}$.

The empty word ε is *compatible* only with synopses (r_0, p_0, q_0) with $r_0 \in \{p_0, q_0\}$. For $u \in \Gamma^*$ and $a \in \Gamma$, the word ua is *compatible* with a synopsis σ of the form (9) if $r_0 \cdot ua \in \{p_\ell, q_\ell\}$ and one of the following holds:

(a) $r_0 \cdot u$ is in the SCC of $r_0 \cdot ua$, and u is compatible with the synopsis obtained from σ by replacing (r_ℓ, p_ℓ, q_ℓ) with $(r_\ell, r_0 \cdot u, r_0 \cdot u)$;
(b) $\ell > 0$, $r_0 \cdot u \in \{p_{\ell-1}, q_{\ell-1}\}$, $a = a_\ell$, and u is compatible with the synopsis obtained from σ by removing the suffix $\xrightarrow{a_\ell} (r_\ell, p_\ell, q_\ell)$;
(c) $\ell > 0$, $r_0 \cdot ua = p_\ell = p_{\ell-1} = q_{\ell-1}$, and ua is compatible with the synopsis obtained from σ by removing the suffix $\xrightarrow{a_\ell} (r_\ell, p_\ell, q_\ell)$.

Note that if some u is compatible with σ and $r_0 \cdot u = p_\ell$, then u is compatible with every synopsis obtained from σ by replacing q_ℓ with some other state; similarly with p_ℓ and q_ℓ swapped.

The states of $\mathcal{B}$ include all synopses for $\mathcal{A}$ and two sink states: all-accepting $\top$ and all-rejecting $\bot$. The simulation invariant is that after processing a proper prefix w of the encoding of the input tree, either $\mathcal{B}$ is in the state $\top$ and $r_0 \cdot \widehat{v}$ is non-rejective for some prefix v of w, or $\mathcal{B}$ is in a synopsis state σ and $\widehat{w}$ is compatible with σ and if the last symbol of w is an opening tag then $p_\ell = q_\ell$.

Let r_0 be the initial state of $\mathcal{A}$. If r_0 is rejective, the initial state of $\mathcal{B}$ is (r_0, r_0, r_0); otherwise, it is $\top$. The invariant clearly holds before the first tag is processed. Let us see how to define transitions from a synopsis state σ of the form (9) to propagate the invariant.

Suppose that an opening tag a is read and let $s = p_\ell \cdot a = q_\ell \cdot a$. If s is not rejective, move to $\top$. If s is rejective and belongs to the SCC of q_ℓ, continue with (r_ℓ, p_ℓ, q_ℓ) replaced with (r_ℓ, s, s) in σ. If s is rejective but does not belong to the SCC of q_ℓ, continue with $\xrightarrow{a} (s, s, s)$ appended to σ. The invariant propagates.

Suppose a closing tag $\bar{a}$ is read. If p_ℓ is not internal, then $p_\ell = q_\ell = r_0$, which is only possible if $\sigma = (r_0, r_0, r_0)$. The automaton $\mathcal{B}$ then moves to $\bot$. Assume that the invariant holds before $\bar{a}$ is processed. Then, $r_0 \cdot \bar{w} = r_0$. Because r_0 is not internal, it follows that w is empty. Hence, $w\bar{a} = \bar{a}$, which is not a prefix of the encoding of any tree, and the state of $\mathcal{B}$ after processing $w\bar{a}$ does not matter. If p_ℓ is internal, we consider four cases depending on whether p_ℓ and q_ℓ are in the same SCC of $\mathcal{A}$, and whether the shape of the synopsis allows backtracking via a transition that originates outside of the SCC of q_ℓ.

Case A: p_ℓ and q_ℓ are in the same SCC X, and either $r_\ell \notin \{p_\ell, q_\ell\}$ or $a \neq a_\ell$ or $p_{\ell-1}$ is not internal; that is, we can only take (backward)

transitions within X. Consider

$$P = \left\{p \in X \;\middle|\; p \cdot a \in \{p_\ell, q_\ell\}\right\} .$$

Because X contains the internal state p_ℓ and the rejective state q_ℓ, all states in X are internal and rejective. The same holds for $P \subseteq X$. Pick any two $p, q \in P$. Because p_ℓ and q_ℓ meet inside X, so do p and q. It follows that p and q meet in q. Hence, (p, q) is a split state, and p and q are almost equivalent. In a minimal automaton there can be at most two different almost equivalent states, so $|P| \leq 2$. If $P = \emptyset$, then $\mathcal{B}$ moves to $\bot$. Otherwise, $P = \{p', q'\}$ for some p' and q', and $\mathcal{B}$ continues, replacing (r_ℓ, p_ℓ, q_ℓ) with (r_ℓ, p', q'). Suppose that the invariant holds before $\bar{a}$ is processed. If it holds by (a), then $r_0 \cdot \widehat{w\bar{a}} \in P = \{p', q'\}$ and $\widehat{w\bar{a}}$ is compatible with the synopsis obtained from σ by replacing (r_ℓ, p_ℓ, q_ℓ) with $(r_\ell, r_0 \cdot \widehat{w\bar{a}}, r_0 \cdot \widehat{w\bar{a}})$. Suppose that the invariant holds by (b). This implies that $r_\ell \in \{p_\ell, q_\ell\}$ and $a = a_\ell$, so it must be the case that p_ℓ is not internal. Then q_ℓ is equal to p_ℓ, so not internal either. By (b), $r_0 \cdot \widehat{w\bar{a}} \in \{p_{\ell-1}, q_{\ell-1}\}$, so it is non-internal too. Consequently, $\widehat{w\bar{a}}$ is the empty word, which is possible only if $w\bar{a}$ is the complete encoding of the input tree. But then the invariant is not required to hold. Finally, the invariant cannot hold by (c), because it would imply that $q_{\ell-1}$ and q_ℓ are in the same SCC, which is forbidden by the definition of synopsis.

Case B: p_ℓ and q_ℓ are in the same SCC X, and also $r_\ell \in \{p_\ell, q_\ell\}$, $a = a_\ell$, and $p_{\ell-1}$ is internal; that is, we can also take (backward) transitions that leave X. Note that this is possible only if $\ell > 0$. Consider again the set $P \subseteq X$ introduced above. If $P = \emptyset$, then $\mathcal{B}$ continues, removing the suffix $\xrightarrow{a_\ell} (r_\ell, p_\ell, q_\ell)$ from the synopsis. In this case, only the condition (b) of the invariant might hold before processing $\bar{a}$, so $\widehat{w\bar{a}}$ is compatible with the modified synopsis, and the invariant propagates. Assume that P is nonempty. Let $p' \in \{p_{\ell-1}, q_{\ell-1}\}$ and $q' \in P$. We know that $p' \cdot a$ and $q' \cdot a$ belong to $\{p_\ell, q_\ell\}$, and that p_ℓ and q_ℓ meet in X, so we also have that p' and q' meet in X. Because $q' \in P \subseteq X$, it follows that p' and q' meet in q'. As $p_{\ell-1}$ is assumed to be internal, so is $q_{\ell-1}$, and consequently also p'. The state q' is rejective because all states in P are. It follows that (p', q') is a split state, so p' and q' are almost equivalent. Because $p' \in \{p_{\ell-1}, q_{\ell-1}\} \subseteq X^{\mathrm{c}}$ and $q' \in P \subseteq X$, we conclude that $p' \neq q'$. Using again the fact that there are at most two different almost equivalent states in every minimal automaton, we get that $p' = p_{\ell-1} = q_{\ell-1}$ and $\{q'\} = P$. The automaton $\mathcal{B}$ continues, replacing (r_ℓ, p_ℓ, q_ℓ) with (r_ℓ, p', q') in the synopsis σ. If the invariant holds before processing $\bar{a}$, then either (a) or (b) holds. If (a) holds, then $r_0 \cdot \widehat{w\bar{a}} = q'$, and the invariant propagates like before. If (b) holds, then $r_0 \cdot \widehat{w\bar{a}} = p' = p_{\ell-1} = q_{\ell-1}$, and after processing $\bar{a}$, (c) will hold.

Case C: q_ℓ is in SCC X but $p_\ell \notin X$, and either $r_\ell \notin \{p_\ell, q_\ell\}$ or $a \neq a_\ell$. We then have $p_\ell = p_{\ell-1} = q_{\ell-1}$. Suppose $p \cdot a = p_\ell$ for some internal p and $q \cdot a = q_\ell$ for some $q \in X$. Then it easily follows that p meets with q in q, and so p and q are almost equivalent. Consequently, $p \cdot a = p_\ell$ and $q \cdot a = q_\ell$ are equal, which is impossible because $p_\ell \notin X$. Thus, p and q cannot both exist.

If p does not exist, $\mathcal{B}$ moves to the state it would take from the synopsis σ' obtained from the current one by replacing (r_ℓ, p_ℓ, q_ℓ) with (r_ℓ, q_ℓ, q_ℓ) in σ. Note that σ' falls into Case A. Suppose that the invariant holds before processing $\bar{a}$. If it is by (a), then $r_0 \cdot \widehat{w} = q_\ell$, so $\widehat{w}$ will also be compatible with σ' and the invariant will propagate as shown in Case A. The invariant cannot hold by (b), because this would imply that $r_\ell \in \{p_\ell, q_\ell\}$ and $a = a_\ell$, and we have assumed the contrary. Suppose that the invariant holds by (c). Then $(r_0 \cdot \widehat{w\bar{a}}) \cdot a = r_0 \cdot \widehat{w} = p_\ell$. But, as we have shown, there are no internal states p such that $p \cdot a = p_\ell$. Hence, $r_0 \cdot \widehat{w\bar{a}}$ is a noninternal state. This is possible only if $\widehat{w\bar{a}}$ is empty. Then, $w\bar{a}$ is the whole encoding of the input tree, and the invariant is not required to hold any more.

If q does not exist, the state is chosen similarly, but this time we obtain σ' by removing the suffix $\xrightarrow{a_\ell} (r_\ell, p_\ell, q_\ell)$ from σ. Note that σ' falls into Case A or Case B: $p_{\ell-1}$ is internal because it is equal to p_ℓ, and $p_{\ell-1}$ and $q_{\ell-1}$ are in the same SCC because they are equal. If the invariant holds before processing $\bar{a}$, then it must be by (c). Then, $\widehat{w}$ will also be compatible with σ', and the invariant will propagate as shown in Cases A and B.

Case D: q_ℓ is in SCC X but $p_\ell \notin X$, and both $r_\ell \in \{p_\ell, q_\ell\}$ and $a = a_\ell$. It then follows that $p_\ell = p_{\ell-1} = q_{\ell-1}$ and $r_\ell = q_\ell$. Consequently, $p_\ell \cdot a = q_\ell$ and, because p_ℓ and q_ℓ are almost equivalent, $q_\ell \cdot a = q_\ell$. Suppose that $p \cdot a = p_\ell$ for some internal state p. Then, we have $p \cdot aa = q_\ell \cdot aa = q_\ell$; that is, p meets with q_ℓ in q_ℓ. Since q_ℓ is rejective, it follows that p and q_ℓ are almost equivalent. But that means that $p_\ell = p \cdot a = q_\ell \cdot a = q_\ell$, which is impossible because $p_\ell \notin X$. Hence, no such p exists. Suppose that $q \cdot a = q_\ell$ for some $q \in X \setminus \{q_\ell\}$. Then $q \cdot a = q_\ell \cdot a = q_\ell$ and it follows that q is almost equivalent to q_ℓ. But this is impossible because together with $p_\ell \notin X$ this would give three different almost equivalent states. Hence, such q also does not exist. We let $\mathcal{B}$ continue with the same synopsis. Suppose that the invariant holds before processing $\bar{a}$. If it is by (a), then $r_0 \cdot \widehat{w\bar{a}} = q_\ell$, because it is the only state in X from which the transition over a leads to $\{p_\ell, q_\ell\}$, and the invariant propagates. If the invariant holds by (b), then $r_0 \cdot \widehat{w\bar{a}} \in \{p_{\ell-1}, q_{\ell-1}\}$, but $p_{\ell-1} = q_{\ell-1} = p_\ell$, so for $\widehat{w\bar{a}}$ and σ we will have (c). Finally, if the invariant holds by (c), it follows that $w\bar{a}$ is the whole encoding of the input tree, like in the first subcase of Case C, and the invariant is not required to hold any more.

This completes the construction of $\mathcal{B}$ and the proof that every run of $\mathcal{B}$ over the encoding of a tree T satisfies the invariant. Directly from the invariant it follows that after reading a prefix wa of $\langle T \rangle$ for a opening tag a, we have $p_\ell = q_\ell = r_0 \cdot \widehat{wa}$. To recognize EL it suffices to enrich the synopsis states of $\mathcal{B}$ with the information about the most recently read tag, and move directly to $\top$ whenever a closing tag $\bar{a}$ is read in a state storing the opening tag a and a synopsis with $p_\ell = q_\ell$ accepting in $\mathcal{A}$. The resulting automaton $\mathcal{B}'$ enters $\top$ in the situation described above or if it encounters a prefix v of the encoding such that $r_0 \cdot v$ is not rejective. In the first case, the automaton $\mathcal{B}'$ has detected a leaf such that the branch leading to it is labelled by a word from L. In the second case, $\mathcal{B}'$ has detected a node such that each branch containing this node is labelled by a word from L. Correctness of $\mathcal{B}'$ follows. □

F PROOFS OF LEMMAS FROM SECTION 3.4

We recall the formulations of the lemmas.

Lemma F.1. *Suppose that $c_1 \approx_{i,j} c_2$. For every word w such that $i \leq \lfloor w \rfloor \leq \lceil w \rceil \leq j$, it holds that $c_1 \cdot w \approx_{i-\|w\|, j-\|w\|} c_2 \cdot w$.*

Proof. It suffices to show the lemma for the case when w is a single letter; the general claim follows by straightforward induction on the length of w. Suppose that $w = a \in \Gamma$. Then, $\lfloor w \rfloor = \lceil w \rceil = 1$. Because $c_1 \approx_{i,j} c_2$ and $i \leq 1 \leq j$, it follows the same transition over a will be taken from c_1 and c_2. After the transition is taken, the absolute thresholds between the three kinds of behaviour of registers listed in the definition of $\approx$ do not change, but because the current depth increases by one, the relative thresholds have to be adjusted. This gives precisely $c_1 \cdot a \approx_{i-1,j-1} c_2 \cdot a$. For $w = \bar{a}$ the argument is entirely analogous. □

Lemma F.2. *Let $c_i = (q_i, d_i, \eta_i)$ with $i \in [1;4]$ be configurations of a depth-register automaton $\mathcal{B}$ and let $y, z \in (\Gamma \cup \bar{\Gamma})^+$ be descending words such that $c_1 \xrightarrow{y} c_2 \xrightarrow{z} c_3 \xrightarrow{y} c_4$. If $\mathrm{img}(\eta_1) \subseteq (-\infty; d_1]$ and $c_1 \sim c_3$, then $\mathrm{img}(\eta_4) \cap (d_1; d_2] = \emptyset$.*

Proof. Because y and z are descending, from $\mathrm{img}(\eta_1) \subseteq (-\infty; d_1]$ it follows that $\mathrm{img}(\eta_3) \subseteq (-\infty; d_3]$. Combining this with $c_1 \sim c_3$, we conclude that from configurations c_1 and c_3 the same sequence of transitions will be taken while processing y. But this implies that if a depth $d \in (d_1; d_2]$ was stored in some register ξ while processing y from c_1, the corresponding depth $d' \in (d_3; d_4]$ will be stored in ξ while processing y from c_3. That is, each depth stored when the first copy of y was processed, is overwritten when the second copy of y is processed. Because $\mathrm{img}(\eta_1) \subseteq (-\infty; d_1]$, and both y and z are descending, there is no other way of putting a value from the segment $(d_1; d_2]$ into registers. □

Lemma F.3. *Let $\mathcal{B}$ be a depth-register automaton with k states and ℓ registers, and let $n \geq k \cdot (\ell + 1)$. For every configuration $c = (q, d, \eta)$ of the automaton $\mathcal{B}$ and every descending or ascending word $x \in (\Gamma \cup \bar{\Gamma})^+$, if*

$$\mathrm{img}(\eta) \cap \left[d + \left\lfloor x^{3 \cdot n!} \right\rfloor ; d + \left\lceil x^{3 \cdot n!} \right\rceil \right] = \emptyset ,$$

then

(1) $c \cdot x^{n!} \sim c \cdot x^{n!} \cdot x^{n!}$; and
(2) $c \cdot x^{n!} \cdot x^{n!} \approx_{\lfloor x^{n!} \rfloor - \|x^{n!}\|, \lceil x^{n!} \rceil - \|x^{n!}\|} c \cdot x^{n!} \cdot x^{n!} \cdot x^{n!}$.

Proof. It is well known that for every deterministic finite automaton $\mathcal{A}$ over $\Gamma \cup \bar{\Gamma}$ with at most n states, $p \cdot w^{n!} = p \cdot w^{n!} \cdot w^{n!}$ for every state p and every word w. To see why this is the case, let us analyze the evolution of the state after processing successive copies of w. Already after processing at most n copies a state will repeat, and because $\mathcal{A}$ is deterministic, we will start looping around a cycle in $\mathcal{A}$. After processing all $n!$ copies we are still on the cycle, of course. After processing any number of copies that is divisible by the length of the cycle (measured in the number of w-steps, not single letters), we return to the same state. Because the length of the cycle is at most n, and $n!$ is divisible by every number between 1 and n, the claim follows.

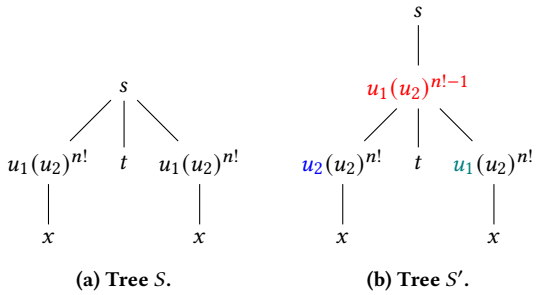
The lemma is proved in a similar fashion. Suppose x is descending; the argument for ascending x is entirely analogous. Throughout the run on $x^{n!} \cdot x^{n!} \cdot x^{n!}$ from c, the current depth stays within $\left[d + \left\lfloor x^{3 \cdot n!} \right\rfloor ; d + \left\lceil x^{3 \cdot n!} \right\rceil \right]$. Consequently, comparisons with values from $\mathrm{img}(\eta)$ give the same result at every step of this run. Moreover, because x is descending, depths stored when processing the ith copy of x are all strictly smaller than every depth that occurs when processing the jth copy of x for all $j > i$. Consequently, the behaviour of $\mathcal{B}$ when processing the $(i+1)$st copy of x is determined by the state and the set of registers storing values not greater than the current depth—after processing the ith copy of x. Because the set of registers can only grow as the successive copies of x are processed, after processing at most $k \cdot (\ell + 1)$ copies of x a state-set pair will repeat. Because the sets only grow, all state-pairs in between share the same set. It follows that when processing subsequent copies of x, this sequence of state-pairs will repeat in a cyclic fashion. Because the length of this sequence is at most $k \cdot (\ell + 1)$, it follows like before that the state-set pairs corresponding to $c \cdot x^{n!}$ and $c \cdot x^{n!} \cdot x^{n!}$ coincide. This implies item (1) of the lemma. In configuration $c \cdot x^{n!} \cdot x^{n!}$ some registers store the same value from

$$\left(-\infty; d + \left\lceil x^{n!} \right\rceil \right] \cup \left(d + \left\lceil x^{3 \cdot n!} \right\rceil ; \infty \right)$$

that they stored in configuration $c \cdot x^{n!}$, and into the remaining registers some values from

$$\left(d + \left\lceil x^{n!} \right\rceil ; d + \left\lceil x^{2 \cdot n!} \right\rceil \right]$$

were loaded when the second copy of $x^{n!}$ was being processed. Because the state-set pairs corresponding to $c \cdot x^{n!}$ and $c \cdot x^{n!} \cdot x^{n!}$ coincide, processing the third copy of $x^{n!}$ will load into the same registers the corresponding (that is, shifted by $\|x^{n!}\|$) values, and no other load operations will be performed. This implies item (2) of the lemma. □



(a) Tree S. **(b) Tree S'.**

Figure 7: Blind variants of fooling trees in Lemma 3.12.

G BLIND CLASSES

We shall use the symbol $\triangleleft$ for the universal closing tag. The *term encoding* $[T] \in \left(\Gamma \cup \{\triangleleft\}\right)^*$ of a tree T with a-labelled root and immediate subtrees $T_1, T_2, \ldots, T_n$ is

$$[T] = a\,[T_1]\,[T_2] \ldots [T_n]\,\triangleleft\ .$$

Correspondingly, for $u = a_1 a_2 \cdots a_n \in \Gamma^*$ we let $\bar{u} = \triangleleft^n$. Like for the markup encoding, we let $[L] = \left\{ [T] \;\middle|\; T \in L \right\}$ for every tree language L over Γ.

A tree language L over Γ is *term-registerless* (resp. *term-stackless*) if there exists a finite automaton (resp. depth-register automaton) over $\Gamma \cup \{\triangleleft\}$ that accepts all words from $[L]$ and rejects all words from $[L^c]$. A unary query Q is *term-registerless* (resp. *term-stackless*) if there exists a finite automaton (resp. depth-register automaton) over $\Gamma \cup \{\triangleleft\}$ that pre-selects nodes in $Q(T)$ when running over $[T]$.

We say that states p and q *blindly meet* in state r if there exist words $u_1, u_2 \in \Gamma^*$ such that $|u_1| = |u_2|$ and $p \cdot u_1 = q \cdot u_2 = r$. By replacing 'meet' with 'blindly meet' in Definitions 3.4, 3.6 and 3.9, we get the definitions of the syntactic classes of *blindly almost-reversible*, *blindly* HAR, *blindly A-flat*, and *blindly E-flat* word languages.

Theorem G.1. *Let L be a regular language.*

(1) EL is a term-registerless tree language iff L is blindly E-flat.
(2) AL is a term-registerless tree language iff L is blindly A-flat.
(3) The following conditions are equivalent:
  (a) QL is a term-registerless unary query;
  (b) EL and AL are term-registerless tree languages;
  (c) L is blindly E-flat and blindly A-flat;
  (d) L is blindly almost-reversible.

Proof. The argument is fully analogous to that in Theorem 3.2, with Lemmas 3.5 and 3.10 to 3.12. replaced by their analogues for term-registerless, blindly E-flat, blindly A-flat, and blindly almost-reversible languages.

The analogue of Lemma 3.5 states that if L is a blindly almost-reversible language, then QL is a term-registerless query. The proof is almost identical, except that when the closing tag $\triangleleft$ is read in state p, we pick any state p' such that $p' \cdot a$ is almost equivalent to p for some $a \in \Gamma$; because L is blindly almost-reversible, the original argument now shows also that the choice of a does not matter.

The analogue of Lemma 3.10 states that a regular language is blindly A-flat iff its complement is blindly E-flat, and that it is blindly almost-reversible iff it is both blindly A-flat and blindly E-flat; it is proved just like the original.

The analogue of Lemma 3.11 states that if L is blindly E-flat, then EL is term-registerless. The proof is an adaptation of the original one to the blind setting. The states of the simulating finite automaton, the simulation invariant, the transitions over opening tags, and the transformation into an automaton recognizing EL are entirely analogous, with 'meet' replaced everywhere with 'blindly meet'; in particular, we keep the labels $a_1, \ldots, a_\ell$ in the synopsis. However, the behaviour of the simulating automaton over the closing tag needs to be adjusted so that it does not rely on the label of the current node. We begin by dropping all references to the current label in the conditions defining Cases A–D, which gives

Case A': $p_\ell, q_\ell \in X$ but either $r_\ell \notin \{p_\ell, q_\ell\}$ or $p_{\ell-1}$ is not internal;
Case B': $p_\ell, q_\ell \in X$, $r_\ell \in \{p_\ell, q_\ell\}$, and $p_{\ell-1}$ is internal;
Case C': $q_\ell \in X$, $p_\ell \notin X$, and $r_\ell \notin \{p_\ell, q_\ell\}$;
Case D': $q_\ell \in X$, $p_\ell \notin X$, and $r_\ell \in \{p_\ell, q_\ell\}$.

In each of these cases the simulating automaton needs to consider all possible values of the current label. That is, in Cases A' and B', the set P is now defined as

$$P = \left\{ p \in X \;\middle|\; p \cdot a \in \{p_\ell, q_\ell\}, a \in \Gamma \right\},$$

and in Case C' we look at $p \cdot a_1 = p_\ell$ and $q \cdot a_2 = q_\ell$ for arbitrary $a_1, a_2 \in \Gamma$. Apart from these differences, the arguments in Cases A'–C' are analogous to the original ones. Let us have a closer look at Case D'. Like before we have $p_\ell = p_{\ell-1} = q_{\ell-1}$ and $r_\ell = q_\ell$. Consequently, $p_\ell \cdot a_\ell = q_\ell$ and, because p_ℓ and q_ℓ are almost equivalent, $q_\ell \cdot a_\ell = q_\ell$. Suppose that $p \cdot a = p_\ell$ for some internal state p and some $a \in \Gamma$. Then, we have $p \cdot aa_\ell = q_\ell \cdot a_\ell a_\ell = q_\ell$; that is, p blindly meets with q_ℓ in q_ℓ. Since q_ℓ is rejective, it follows from blind E-flatness that p and q_ℓ are almost equivalent. Consequently, $q_\ell \cdot a = p \cdot a = p_\ell$. Because we also have that $p_\ell \cdot a_\ell = q_\ell$, it follows that $p_\ell \in X$ which is a contradiction. Hence, such p cannot exist. One then argues, like in the markup case, that there is no $q \in X \setminus \{q_\ell\}$ for which there exists $a \in \Gamma$ such that $q \cdot a = q_\ell$, and that letting the simulating automaton continue with the same synopsis preserves the invariant.

Finally, the analogue of Lemma 3.12 states that for each regular language L, if EL is term-registerless, then L is blindly E-flat. This time there are important differences in the proof; we sketch it below.

We show that if L is not blindly E-flat, then $[EL]$ cannot be separated from $[(EL)^c]$ by a finite automaton. Suppose that the minimal automaton $\mathcal{A}$ of $L \subseteq \Gamma^*$ is not E-flat. Let i be the initial state of $\mathcal{A}$. Then, there exist words $s, t, u_1, u_2 \in \Gamma^+$, $x \in \Gamma^*$ and states p, q such that $|u_1| = |u_2|$, $i \cdot s = p$, $p \cdot u_1 = q \cdot u_2 = q$, $q \cdot x$ is rejecting, and $p \cdot t$ is accepting iff $q \cdot t$ is rejecting. It follows that for each $k > 0$, $su_1(u_2)^k x \in L^c$, and $st \in L$ iff $s(u_1)(u_2)^k t \in L^c$. Unlike for the markup encoding, the construction of the fooling trees depends on whether $st \in L$ or $st \in L^c$

Suppose first that $st \in L^c$. Then, the trees S, S' used in Lemma 3.12 should be replaced with the ones in Fig. 7a. We have $S \notin EL$ and $S' \in EL$. Note that we have no control on whether the rightmost branch of S' is labelled by a word from L or not, but it is irrelevant, because we know that the middle branch is. The term encodings of S and S' satisfy the following:

$$\begin{aligned}
[S] &= s \cdot u_1(u_2)^{n!}\, x\bar{x}\, (\bar{u}_2)^{n!}\bar{u}_1\, t\bar{t}\, u_1(u_2)^{n!}\, x\bar{x}\, (\bar{u}_2)^{n!}\bar{u}_1\, \bar{s}\,, \\
[S'] &= s\, u_1(u_2)^{2\cdot n!}\, x\bar{x}\, (\bar{u}_2)^{n!}\bar{u}_2\, t\bar{t}\, u_1(u_2)^{n!}\, x\bar{x}\, (\bar{u}_2)^{n!}\bar{u}_1(\bar{u}_2)^{n!-1}\bar{u}_1\, \bar{s} \\
&= s\, u_1(u_2)^{2\cdot n!}\, x\bar{x}\, (\bar{u}_2)^{n!}\bar{u}_1\, t\bar{t}\, u_1(u_2)^{n!}\, x\bar{x}\, (\bar{u}_2)^{n!}\bar{u}_2(\bar{u}_2)^{n!-1}\bar{u}_1\, \bar{s} \\
&= s\, u_1(u_2)^{2\cdot n!}\, x\bar{x}\, (\bar{u}_2)^{n!}\bar{u}_1\, t\bar{t}\, u_1(u_2)^{n!}\, x\bar{x}\, (\bar{u}_2)^{2\cdot n!}\bar{u}_1\, \bar{s}\,,
\end{aligned}$$

because $|u_1| = |u_2|$ implies $\bar{u}_1 = \bar{u}_2$. The rest of the proof is identical.

If $st \in L$, in S we replace u_1 on the rightmost branch with u_2, and we modify S' accordingly. It then holds that $S \in EL$ regardless of whether $su_2(u_2)^{n!}x$ belongs to L or not, and $S' \notin EL$; the proof again continues like in Lemma 3.12. □

Theorem G.2. *For each regular language L, the following conditions are equivalent:*

(1) QL is a term-stackless unary query;
(2) EL is a term-stackless tree language;
(3) AL is a term-stackless tree language;
(4) L is blindly HAR.

Proof. The argument is fully analogous to that in Theorem 3.1, with Lemmas 3.7, 3.8 and 3.16 replaced by their analogues for term-stackless and blindly HAR languages.

The analogue of Lemma 3.7 states that the class of blindly HAR languages is closed under complement, which is immediate from the definition just like for HAR languages.

The analogue of Lemma 3.8 states that if L blindly HAR then QL is term-stackless. The proof is analogous, with the only modification being what we did with Lemma 3.5 in the proof of Theorem G.1: when the closing tag $\triangleleft$ is read in state p and the current depth is greater than or equal to the maximal stored depth, we pick any

state p' such that $p' \cdot a$ is almost equivalent to p for some $a \in \Gamma$. Because L is blindly HAR, the original argument now shows also that the choice of a does not matter.

Finally, the analogue of Lemma 3.16 states that for each regular language L, if EL is a term-stackless tree language then L is blindly HAR. The proof is obtained by adjusting the proof of Lemma 3.16 just like the proof of Lemma 3.12 was adjusted in Theorem G.1. This time there is only one case because we know that $s(wu_1+vu_2)^*wt \subseteq L^c$ and $s(wu_1 + vu_2)^*vt \subseteq L$, and not the other way around. In the tree R shown in Fig. 5, the copies of u immediately following copies of w should be replaced by u_1 and those immediately following v should be replaced by u_2. From there, the proof continues like before. □